APPLICATIONS OF MULTI-THREADING PARADIGMS TO SIMULATE TURBULENT FLOWS

Igor Grossman

College of Engineering and Science
Footscray Park Campus, Victoria University
Melbourne, Australia

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

ABSTRACT

Flow structures in turbulent flows span many orders of magnitude of length and time scales. They range from the length scale at which very small eddies lose their coherence as their translational kinetic energy is dissipated into heat, up to eddies the size of which is related to that of the macroscopic system. The behaviour of the range of flow structures can be captured by assuming that the fluid is a continuum, and they can be described by solving the Navier-Stokes equations. However, analytical solutions of the Navier-Stokes equations exist only for simple cases.

A complete description of turbulent flow in which the flow variables velocity and pressure are resolved as a function of space and time can be obtained only numerically. The instantaneous range of scales in turbulent flows increases rapidly with the Reynolds number. As a result, most engineering problems have a wide range of scales that can be computed using direct numerical simulation (DNS). As the complexity of the calculated flows increases, an improvement in turbulence models is often needed. One way to overcome this problem is to search for models that better capture the features of turbulence.

Furthermore, the models should be parameterised in a way that allows flows to be simulated under a wide range of conditions. DNS is a useful tool in this endeavour, and it can be used to complement the long-established methodologies of experimental research. A large number of computational grids must be used to simulate a high Reynolds number inflows that occur in the complicated geometries often encountered in practical applications. This approach requires a considerable amount of computational power. For example, reducing the grid spacing in half increases the computational cost by a factor of about sixteen. Challenges presented by limitations imposed by computer hardware significantly limit the number of practical numerical solutions required to satisfy engineering needs.

In this work, we propose an alternative approach. Rather than running an application that solves the Navier-Stokes equations on one computer, we have developed a platform that

allows a group of computers to communicate with one another working together to obtain a solution of a specific flow problem.

This approach helps to overcome the problem of hardware limitations. However, to grasp these challenges, we must devise new strategies to computational paradigms associated with parallel computing. In the case of solving the Navier-Stokes equations, we have to deal with significant computational and memory requirements. To overcome these requirements, software should be able to be run on many high-performance computers simultaneously, and network communication may become a new limiting issue that is specific only to parallel environments. Translating to parallel environments triggers several scenarios that do not exist when developing software that executes sequential operations. For example, "racing conditions" may appear that result in many threads that attempt to use different values of a shared variable, or they simultaneously attempt to overwrite it. The order of executions may be random as the operating system can swap between the threads at any time. Attempts to synchronise the threads may result in "deadlock" when all resources become simultaneously locked. Debugging and problem-solving in parallel environments is quite often difficult due to the potentially random nature of the orders in which threads run. All of these features require the development of new paradigms, and we must transform our way of envisioning the development of software for parallel execution. The solution to this problem is the motivation for the work presented in this thesis.

A significant contribution of this work is to strategically use the ideas of thread injection to speed up the execution of sequential code. Bottlenecks are identified, and thread injection is used to parallelise the code that may be distributed to many different systems. This approach is implemented by creating a class that takes control over the sequential instructions that create the bottlenecks. The challenge to engineers and scientists is to determine how a given task can be split into components that can be run in parallel. The method is illustrated by applying it to *Channelflow* (Gibson, 2014), which is open-source Direct Numerical Simulation software used to simulate flows between two parallel plates.

Another challenge that arises when approaching representations of real geometries is the scale and magnitude of the data samples. For example, Johns Hopkins Turbulent Database

(JHTB) contains results of the solution of direct numerical simulation (DNS) of isotropic turbulent flow in the incompressible fluid in 3D space and only requires 100 TB data. Much more data needed to perform a simulation, and this is just a straightforward model.

A natural answer to this challenge is to exploit the opportunities offered by contemporary applications of 'database technology' in computational fluid dynamics (CFD) and turbulence research. Direct numerical solution of the Navier-Stokes equations resolves all of the flow structures that influence turbulent flows. Still, in the case of Large Eddy Simulation, the Navier-Stokes equations are spatially filtered so that they are expressed in terms of the velocities of larger-scale structures. The rate of viscous dissipation is quantified by modelling the shear stress, and this process can lead to inaccuracies. A means of rapid testing and evaluation of models is therefore required, and this involves working with large data sets.

The contribution of this work is the development of a computational platform that allows LES models to be dynamically loaded and to be rapidly evaluated against DNS data. An idea permeating the methodology is that a core is defined that contains the 'know-how' associated with accessing and manipulating data, and which operates independently of a plug-in. The thesis presents an example that demonstrates how users can examine the accuracy of LES models and obtain results almost instantaneously. Such methods allow engineers or scientists to propose their own LES models and implement them as a plug-in with only a few lines of code. We have demonstrated how it can be done by converting the Smagorinsky model to a plug-in to be used on our platform.

ACKNOWLEDGMENTS

After four years spent at Victoria University doing my PhD, I have quite a few people to thank for the help and the support I received. However, it is quite hard to recall everybody when looking back over this period. Therefore, my apologies to those I may have forgotten in the following; these acknowledgments are for you too.

I want to express my deepest gratitude to my supervisor, Professor Graham Thorpe, not just for the opportunity he gave me but also for patiently guiding me throughout my whole research path.

As well, I would like to thank Professor Jun De Li. We had many discussions at the beginning of the project. He has helped and supported me, sharing all the difficulties I had to face while fighting with the code.

Also, I want to thank Professor Vasilij Novozhilov. I have met him in person just twice but the discussions I had with him and his willingness to help have been beneficial and much appreciated.

I wish to thank Victoria University for the opportunity to access the fastest Melbournian supercomputer at the time.

In the last four years, I have collaborated with many people within the Edward and Spartan supercomputer group at Melbourne University. I want to acknowledge and thank all of them, but special thanks to Lev Lafayette, who has been the first point of contact when something went wrong at the supercomputer.

Finally, I want to thank my family and in particular my wife Rita for her patience over this entire path.

DECLARATION

I, Igor Grossman, declare that the PhD thesis entitled APPLICATIONS **OF MULTI-THREADING PARADIGMS TO SIMULATE TURBULENT FLOWS** is no more than 100,000 words in length including quotes and exclusive of tables, figures, appendices, bibliography, references, and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work.

Signature: 

Date: 21/12/2017

Table of Contents

PAPERS

The context of this thesis is based on the following papers which have been published or been accepted for publication during this PhD project:

1. First Thermal and Fluids Engineering Summer Conference, 9-12 August 2015, New York, NY, USA
   https://www.astfe.org/conferences/tfesc/TFESC_Conference_Technical_Program.pdf

2. Begell House, A PLATFORM THAT ACCEPTS SUB-GRID MODELS AS PLUGINS TO ENABLE THE TESTING OF LES MODELS AGAINST DNS DATA,
   http://search.begellhouse.com/index.php

3. Thread injection to make DNS Channelflow run in parallel:
   Part1, http://vuir.vu.edu.au/31080/

4. Thread injection to make DNS Channelflow run in parallel:
   Part2, http://vuir.vu.edu.au/31079/

5. A platform that accepts sub-grid models as plugins to enable the testing of LES models against DNS data archived in the John Hopkins database. http://vuir.vu.edu.au/30976/

1 THE POWER OF PARALLEL COMPUTING APPLIED TO COMPUTATIONAL FLUID DYNAMICS

1.1 INTRODUCTION

The flow of fluids pervades our very existence (Tennekes and Lumley, 1972). Blood moves through our body; air flows in our lungs - even cosmic dust clouds manifest turbulent-like behaviour (Yang *et al*. 2015) as they approach black holes. Turbulent flow is virtually everywhere. Indeed, many of the environmental and energy-related issues we face today cannot be resolved without detailed knowledge of the mechanics of turbulent flows. But what exactly is turbulence?

Turbulence is a flow composed of eddies or vortices: patches of often swirling fluid, moving randomly about the overall direction of motion. Technically, the chaotic state of fluid motion arises when the speed of the fluid exceeds a specific threshold, below which viscous forces damp out the chaotic behaviour (Tritton, 1988).

Perhaps the simplest way to define turbulence is to invoke the Reynolds number, a parameter that compactly characterises a flow. The magnitude of the Reynolds number indicates the ratio of inertial to viscous forces that arise as a result of fluid flow (Smagorinsky, 1963). If we list the flows that capture the attention of most scientists and engineers, we will find that practically all of them are turbulent. Turbulence is the rule, not the exception, in the behaviour of fluids. The flow of fluids is governed by the Navier-Stokes equations, which are derived from the principle of the conservation of momentum. One of the terms in the equation is non-linear in velocity, and this accounts for the formation of instabilities that can occur within fluid flows.

Nobel Laureate Richard Feynman was moved to declare turbulence to be "the most important unsolved problem of classical physics" (Feynman et *al*., 1963). The question can be invited – how do engineers deal with problems involving turbulent flows if the nature of turbulence is still considered an unsolved problem? Fundamentally, they use the best science available to them at the time.

1.2 AN ENGINEERING APPROACH

Engineering projects generally pass through several stages. They begin with preliminary research during which the project requirements need to be specified. Next, the requirements of the project are addressed, and engineers produce some possible solutions and select the one they believe will best suit their needs. When performing calculations, engineers generally draw on contemporary science. Still, often this comprises little more than simplified equations, a sort of "recipe" or "cookbook", empirical formulae and an endless number of parameters reflecting generations of accumulated data and experience. After this stage, development can begin with the introduction of a working prototype.

To arrive at this point, a great deal of money has generally been invested – and this is only the first iteration. The question arises – is there any other way? This question can be answered in two words - computer simulation (Pope, 2004). If we condensed all the time that the study of turbulent flows has been carried out to the duration of one week, the most significant advances in our practical results would have taken place in the last hour. A result of this is the introduction of computer simulation and the emergence of supercomputing (Sanders *et al*., 2011).

An example is a critical role that supercomputers have played in the success of biomedical science. In 1999 IBM announced a $100 million dollar initiative to build the petaflop-scale supercomputer to tackle the protein folding problem. The IBM Blue Gene project was targeting massive parallel simulations of biomolecular systems to advance our knowledge in the understanding of biological processes.

1.3 MATHEMATICAL MODELING OF TURBULENT FLOWS

The governing equations for laminar, transition and incompressible turbulent flows are the Navier-Stokes equations

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = f_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x}\left(\nu \frac{\partial u_i}{\partial x_j}\right), \tag{1.1}$$

complemented with the mass conservation and incompressibility constraint

$$\frac{\partial u_i}{\partial x_i} = 0, \tag{1.2}$$

where u_i is the velocity component in the x_i direction, f_i represents external forces, p pressure and ν is the kinematic viscosity of the fluid. These equations were derived independently almost two centuries ago by the French engineer Claude Navier and the Irish mathematician George Stokes. They are equations which govern the velocity and pressure of fluid throughout a flow field. The computational difficulty arises principally from the non-linear term, $u_j \frac{\partial u_i}{\partial x_j}$, which is ultimately responsible for the growth in the number of lengths and time scales as a flow undergoes a transition from laminar to turbulent flow. The evaluation of this term is computationally resource-intensive. In this work, we shall enter the realm of parallel computing with its unique paradigms and logic to expedite the computation of this non-linear term.

1.4 NUMERICAL SIMULATION

The Navier-Stokes equations are strongly non-linear, and mathematically they cannot be readily solved when the Reynolds number is high, or the domain in which the fluid occupies is complex. It is only recently that fluid dynamicists have been able to solve them approximately by making

use of computers. As a result, a new sub-branch of classical physics, namely computational fluid dynamics, was born.

Geometrical representation, along with computer visualisation, has transformed how the results of CFD are portrayed and interpreted. This is a complex task, bringing together mathematicians, computational scientists and engineers. Generally speaking, it is required to find the numerical "recipe" to split the geometry into several smaller entities of simple shapes which constitute a computational spatial mesh. The success of commercial CFD packages depends in part on the speed, accuracy, and reliability in which this can be done. Having developed a suitable mesh generator, the differential equations that govern the flow are discretised on the mesh and solved by advancing the solution in finite time steps.

Currently, there are three different levels of approximation used to simulate turbulent flows using computers. They range from the most detailed, refined and accurate solutions through to those that embody sweeping approximations. They are:

- Direct numerical simulation (DNS) (Krist and Zang, 1987)

  In DNS, the Navier-Stokes equations (1.1-1.2) are solved numerically by resolving the shortest time and spatial scales of the flow field. The principal application of DNS is to help establish the fundamentals of turbulence. As a result, much of the literature about DNS is carried out in simple geometries as exemplified by the work of Reveillon *et al*. (2011). Furthermore, DNS can only simulate flows at relatively low Reynolds numbers because of the limitations of current computer power in terms of computation speed and memory.

- Large-eddy simulation (LES) (Smagorinsky, 1963).
  Large Eddy Simulation has features that are akin to both Reynolds averaged and direct numerical simulation methods. It solves the unsteady partial differential equations that account for the conservation of mass, momentum, and energy at the large scales of motion and the small eddies are modelled.
- Reynolds averaged Navier-Stokes (RANS) (Reynolds, 1895) computational fluid dynamics (CFD). The Reynolds-averaged or RANS equations usually are time-

averaged equations of motion that govern fluid flow. The idea behind these equations is the partitioning of the instantaneous flow field into the sum of time-averaged and time-fluctuating components that are deemed to influence the flow.

1.4.1 DIRECT NUMERICAL SIMULATION (DNS)

The Navier-Stokes equations have been established for almost 200 years. Except in the cases of a few simple flows Muriel, (2010), Muriel and Dresden, (1997), no analytical solutions have been obtained. Flow structures in turbulent flows span many orders of magnitude of length and time scales. They range from the length scale at which very small eddies lose their coherence as their translational kinetic energy is dissipated into heat, up to eddies the size of which are related to that of the macroscopic system. The behaviour of the range of flow structures can be captured by assuming that the fluid is a continuum, and they can be described by solving the Navier-Stokes equations. Richardson (1961), who introduced point iterative schemes for numerically solving Laplace's equation, is regarded as the progenitor of computational fluid dynamics and his concept of turbulence endures to the present time. He is well known for the ditty he composed that captures the nature of energy cascade in turbulent flows, namely

Big whorls have little whorls that feed on their velocity,
and little whorls have lesser whorls and so on to viscosity.

The underlying idea is that those turbulent flows are composed of ‘eddies’ of different sizes. Increasing the Reynolds number leads to the activation of smaller turbulent flow scales down to a lower limit. The smallest scale is known as the Kolmogorov scale, and it needs to be resolved in the most detailed of numerical simulations. The accuracy of the solutions is strongly dependent on the spatial and temporal resolutions employed.

Direct numerical simulation (DNS) is simulation when results of numerically solving Navier-Stokes equations are achieved without any turbulence model. This will be thought of as all properties of turbulent flow can be retrieved based on smallest time and smallest space intervals. The requirement of the mesh size can be determined by the Kolmogorov scale and given by

$$\eta = (\nu^3/\varepsilon)^{1/4} \tag{1.3}$$

where ν is the kinematic viscosity, and ε is the rate of kinetic energy dissipation. The integral scale depends on a spatial scale for given boundary conditions. To satisfy this requirement, several node points for the mesh must maintain the integral scale in the range of the computational domain

$$Nh > L \tag{1.4}$$

where N is a number of points in mesh direction and h is a space step size.
All the above make step h is to follow equation (1.5)

$$h \leq \eta \tag{1.5}$$

And because

$$\varepsilon \cong u'^3/L \tag{1.6}$$

where u' is the root mean square of the velocity and for three-dimensional space, the number of mesh points should satisfy

$$N^3 \geq Re^{9/4} \tag{1.7}$$

and Re is turbulent Reynolds number:

$$Re = \frac{u'L}{\nu} \tag{1.8}$$

all the above equations conclude that the storage requirement is growing very fast when we try to simulate flows with high Reynolds numbers. Also, the time step to produce accurate results should be correspondingly small. All that makes DNS very expensive, even for most powerful computers. DNS is mainly suitable for numerical experiments but

can hardly be used for practical engineering tasks.

REYNOLDS-AVERAGED NAVIER-STOKES EQUATIONS

One of the approaches to simplify Navier-Stokes equations of motion of the fluid flow is to formulate Reynolds averaged or (RANS) equations. They are time-averaged equations of motion for turbulent flow and can be expressed as:

$$\rho\bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = \rho\bar{f}_i + \frac{\partial}{\partial x_j}\left[-\bar{p}\,\delta_{ij} + \mu\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \rho\overline{u_i' u_j'}\right] \quad (1.9)$$

The idea behind the Reynolds-averaged approach is separating velocity into two components, namely a mean (time-averaging) component, and a fluctuating component as follows

$$u(x,y,z,t) = \overline{u(x,y,z)} + u'(x,y,z,t) \quad (1.10)$$

This approach was initiated by Reynolds (1895).

1.4.3 LARGE-EDDY SIMULATION (LES)

Large-eddy simulation (LES) is a three-dimensional unsteady simulation procedure that attempts to capture the physics of turbulence. In LES, turbulence contained in large length scales is resolved, and small-scale turbulence is modelled (You & Moin 2007). The resolved scales are obtained by solving the Navier-Stokes equations directly, and this allows the temporal and spatial evolution of those eddies to be captured. However, modelling is required for the effects of dissipation of kinetic energy in the unresolved small-scale eddies (Yuan & Piomelli 2015, You & Moin 2007). These models are known as the sub-grid scale (SGS) models. LES partitions the large and small-scale eddy motions in physical space by using a filtering technique. Eddies smaller than the filter width are modelled by the filtering process, which reduces the computational cost compared to direct numerical simulation (DNS) (You & Moin 2007). Traditional LES employs implicit filtering schemes in which the filter width usually is the same as the grid size. This effectively means that if the grid changes, the model changes with the grid size, which is likely to give different solutions.

Moreover, there is less control over the rise of numerical errors due to direct dependency on the grid size (Gnanaskandan & Mahesh 2016). Thus, grid independence becomes elusive using implicit LES (Sarwar et al., 2017). Unless a grid-independent result is obtained, questions will remain about the numerical accuracy of a CFD simulation.

In LES explicit filters apply to the discretised Navier-Stokes equation with well-defined filter shapes. The approach is based on means of controlling numerical errors that result when finite-different methods are used and operation that reduce truncating errors. However, using explicit filtering requires a much higher density grid and computational costs increase with $(\Delta x)^4$

One of the central tenets of LES is Kolmogorov's theory of "self-similarity" which allows one to separate small and large eddies. The large eddies of the flow are dependent on the geometry of the domain while the minor scales are more or less universal. Large-eddy motions are retained and obtained directly using a transient calculation. Large-eddy simulations are inherently approximate because the effects on the flow field of the small eddies are based on heuristically formulated models. We shall show in Chapter 5 a computationally efficient way of establishing the accuracy of LES models by comparing them with data generated by DNS. However, the flow fields associated with DNS require the manipulation of perhaps petabytes of data, and it is one of the aims of this thesis to develop powerful computational methods of efficiently analysing such large data sets.

In LES, a spatial filtering operation using a kernel *G* is applied to the flow field as follows:

$$\bar{\varphi} = \int G(\vec{x} - \vec{y})\, \varphi(\vec{y}) \mathrm{d}\vec{y} \tag{1.11}$$

resulting in one being able to express a flow variable, φ, as the sum of two components thus

$$\varphi = \bar{\varphi} + \tilde{\varphi} \tag{1,12}$$

where $\bar{\varphi}$ is the resolvable scale component and $\tilde{\varphi}$ is a sub-grid-scale component.

Using the decompositions $u_i = \bar{u}_i + \tilde{u}_i$ and $p = \bar{p} + \tilde{p}$ and applying the filtering operation to the Navier-Stokes and continuity equations, we obtain

$$\rho\left(\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j}\right) = \rho \bar{f}_i - \frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\mu \frac{\partial \bar{u}_i}{\partial x_j}\right) + \frac{\partial \tau_{ij}}{\partial x_i} \tag{1.13}$$

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0 \tag{1.14}$$

wherein (1.13),

$$\tau_{ij} = \overline{u_i u_j} - \bar{u}_i \bar{u}_j \tag{1.15}$$

The term τ_{ij} in (1.15) captures the residual stresses which need to be modelled using suitable sub-grid-scale methods. Equation 1.15 gives rise to one of the central challenges of LES, namely, how do we accurately model the residual stresses? The statistical properties of turbulent flows can be accurately determined by making use of DNS, and the latter can provide benchmark solutions against which LES can be evaluated. However, because DNS solutions are data-intensive, engineers and scientists require computationally efficient methods of handling large amounts of data. As part of this research, we have developed an easy-to-use platform that enables a range of LES models to be evaluated against DNS data.

1.4.4 COMPUTATIONAL CHALLENGES

It is somewhat trite to claim that experiments are expensive and time-consuming to carry out and that these difficulties can be obviated by performing numerical experiments. However, there is a serious impediment to simulating many of the systems that arise in practice, namely the inability of computers to solve the vast number of simultaneous equations that govern the flow of fluids. Because of the economic value and safety requirements imposed on the design of aircraft, it appears that aircraft manufacturers have supported a considerable amount of experimental and computational fluid dynamics research. The research presented in this thesis contributes know-how to the requirements to adapt computer code to HPC architectures, and the need for improved

physical modelling. This is closely aligned with the directions of future CFD research outlined by Kroll *et al*. (2015). For example, aircraft must decrease their speed as they are landing, and this has the potential to reduce the lift that is essential in keeping them airborne. Aircraft designers, therefore, resort to modifying the shape of the airfoils that constitute the wings employing a complex array of flaps and slats that control the flow to prevent the streamlines from separating from the surfaces of the wings which result in a loss of lift. Many details of the flow can be resolved using LES, but if the behaviour of the complete airframe is to be determined, about one trillion grid cells must be used (Chapman, 1979). This is but one example where numerical experiments are challenging to carry out. For example, Jameson points out that on a petaflop computer, a DNS solution of flow around an A380 would take about 30 years. To correctly simulate turbulent flow the grid must be significantly refined in the direction of the wall. This makes for Re>O (10^5), over 90% of the grid points are needed to resolve less than 10% of the computational domain. Without a doubt, the requirement for simulation is quite high – the number of the grid point is proportional to $Re^{9/4}$ and the overall cost is proportional to Re^3. According to Moore's law (Moore *et al*. 1965), the density of transistors will double every two years. However, when we are getting closer to the atomic scale increase, the computer power will cease, and we are approaching this limit.

The fact is this: we do not expect the one-trillion cell memory to be available soon (Kroll *et al.* 2015). With a petaflop computer (IBM Roadrunner, 2008), the DNS of the A380 would take about 10^9seconds –-about 30 Years!

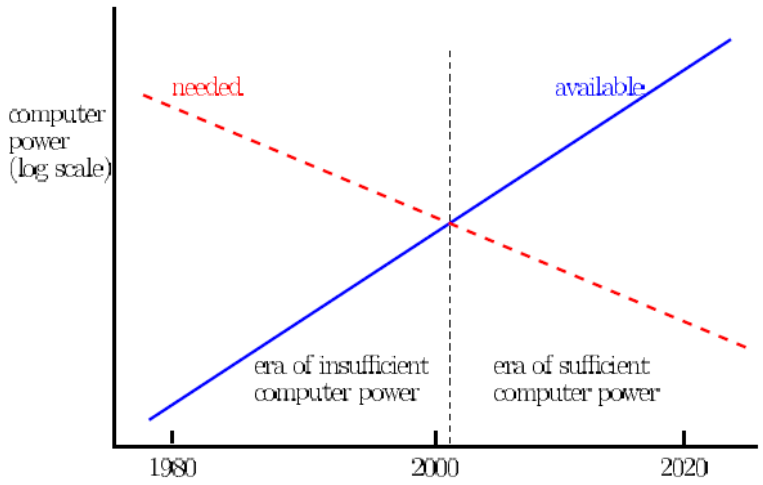


Figure 1.1 Sketch of the computer power available and that needed for LES as a function of time. The cross-over time is the transition from the era of insufficient computer power to the era of sufficient computer power. (Pope, 2004)

Figure 1.1 shows the relation of power available and needed for LES. It is quite evident that until we have enough computational capabilities, simulation can be done only for the simple flows and simplifications are unavoidable. However, when we come to the second era of significant power, we expect that it will grow not only in quantity but also in quality. This will occur when computer power starts to grow via computers joining in one global network. So, one complicated flow simulation should be viable if solved by many computers.

This research approach is to remove the impediment that prevents significant, practical problems being solved by developing new approaches to the design of the software.

1.4.5 PARALLELISATION

Two principal factors are driving the parallelisation of computer hardware and software. The rate of acceleration of the speed of processors is slowing (Lavington, 1998; Norberg and O'Neil, 1996). This thesis is concerned with solving the Navier-Stoke equation, the fundamental equation that governs how flowing fluids behave. Parallelisation is the key to making further progress in this area of research, and it will also help us to develop accurate, but computationally efficient methods of modelling flow in fluids. This latter consideration is also addressed in this work.

The conventional development of software was based on serial programming Scott (2009). In this traditional approach, algorithms were assembled into discrete instructions, each of which is executed one by one by a central processor. Only when one such instruction is finished will the execution of the next instruction be initiated. In the meantime, all of the computer resources are waiting. This represents a considerable waste of resources

Although serial processing has severe limitations, there is no doubt that over the three or so decades-spanning 1970 – 2004 developments in CPUs resulted in significant increases in processing speed. For example, Sutter (2005) has quantified the trend in Intel CPU development and Figure 1 illustrates the increase in performance of Pentium® personal computer chips.

Examining this graph and comparing CPU performance for Pentium Personal Computers, we can acknowledge that regardless of software quality, any new generation of hardware will significantly speed up any application. Progress occurred at a high pace as new generations of hardware appeared on personal computers. However, it can be seen from Figure1.2 that the rate of increase in speed began to decrease around 2005.

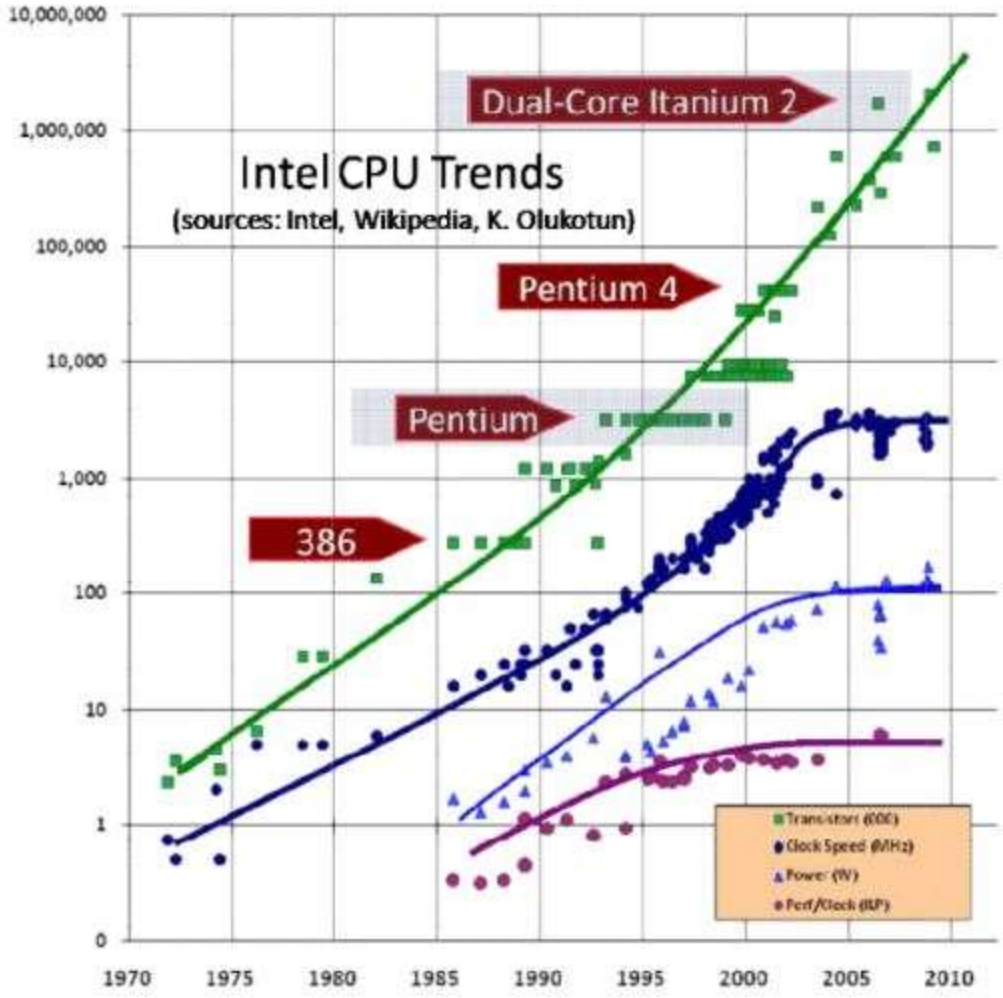


Figure 1.2 The trend in Intel CPU development illustrates that the increase in the speed of computer chips decreases after about 2005. The time in years is plotted on the abscissa and the number of transistors, clock speed, power consumption and the number of computer instructions per second are plotted on the ordinate. The graph was updated in August 2009, but the original source is Sutter (2005).

Similar results were garnered by McCalpin *et al.* (2011), for other chips, suggesting that the increase in performance of single-threaded CPU performance has begun to decline.

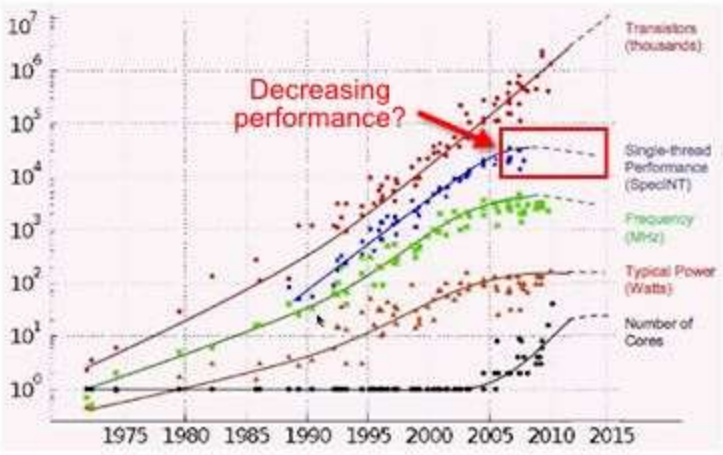


Figure 1.3 These data presented by McCalpin *et al.* (2011) suggest that the rate of increase in chip performance is slowing.

The overarching message portrayed in the figures is that in around the ’80s and 90’s virtually any software ran faster in terms of doubling its speed in around 18-20 months. However, eventually, the growth in performance of single-threaded CPUs began to slow, and it appeared to hit a limit (Sutter, 2005). By 2005 the increase in the speed of single-threaded software plateaued. That was a time when the focus turned to the development of parallel software.

According to Culler *et al*. (1999), there are three types of parallelism, namely

- **Bit-level parallelism:** This is the form of parallel computing which is based on increasing the word size of the processor. Increasing the word size usually results in reducing the number of instructions that the system must execute to perform a given task.
- **Instruction-level parallelism:** At run time the processor is dynamically ordered to process instructions in parallel
- **Task Parallelism:** Task parallelism employs the decomposition of a task into subtasks and then allocates each of the subtasks for execution. The processors perform execution of sub-tasks concurrently.

When bit-level parallelism and instruction-level parallelism are performed automatically by the hardware and operating system, we have already noted that this way of the increasing power of computer calculations reached its apogee in 2004.

Task parallelism requires direct intervention by software developers if the performance of computers is to be advanced, and this is at the core of the research presented in this thesis. Developments in this direction, initiated by Gropp *et al.* (2001), gave birth to new constraints on how computer software is conceived and written, especially where the underlying computer architecture been directly accessed through the application programming interface (API).
A significant step in parallel software development was made by Hollman (2016), who created the pThread library for C++. This add-on enabled software developers to use sequential C++ language to develop parallel code using API located in the pThread library and that incorporated 100 functions. Subsequently, parallel concepts were directly encapsulated at the software language level. For example, from the time of its inception Java (Frumkin, 2003) incorporates a runnable thread interface.

Eventually, new languages like CUDA (Nvidia, 2007) were created to target massive parallel software development especially. CUDA has been bound to many software languages such as FORTRAN, IDL, Java, MATLAB, Mathematica, Pyton, .Net, and so on. A significant difference in parallel software development compared to sequential programming is a necessity to split the underlying task into several independent subtasks, each of which can be executed concurrently. This makes parallel development very targeted to the type of application.
There are some tasks which are naturally parallel, like rendering in computer visualisation Eilemann (2019), brute force searches Loesch (1990) and so on. Moler (1986) coined the phrase "embarrassingly parallel" to describe problems that are definitively parallel and easy to solve. In contrast, performing parallelisation in CFD applications is quite challenging; Simon (1991)**.** However, the growing trend of publications on CFD Singh *et al.* (2018) shows the significant interest of researchers in this area.

At the International parallel CFD conference held in 2018, a number of the new directions was presented to attack problems of parallelisation. They included:

- Parallel Algorithms and Solvers
- Extreme-Scale Computing
- Mechanical and Aerospace Engineering Applications

- Atmospheric and Ocean Modelling
- Medical and Biological Applications
- Fluid-Structure Interactions
- Turbulence and Combustion
- Acoustics
- Software Frameworks and GPU Computing

The fact that the implementation of parallel CFD has given rise to so many strands of research and applications is a clear demonstration of the intensity of the research effort in this area. The discovery of reusable independent parts with the ability to perform parallel CFD provokes many new questions such as:

Large scale CFD is based on several unstructured domains, and the question arises regarding how to distribute this large number of unstructured domains over several processors with distributed memory, and how to achieve a balance of load and what the optimum number of processors required. This question was addressed by Simon (1991) and an algorithm using a graph of the theoretical framework with three decompositions was introduced. The authors show that the computation of an eigenvector of Laplacian matrix associated with a graph gets superior results for this spectral bisection algorithm, and it leads to the solution of distribution of unstructured domains through the number of processors and achieving a balance of load.

Another question on how to dynamically partition unstructured meshes in a parallel way was addressed by Walshaw *et al.* (1997) who introduced an iterative gain optimisation technique to archive load balancing and minimised inter-process communication overhead. Their experiments show that adaptively refined meshes produced similar or higher quality partitioning and much more rapidly than a sequential approach.

Solver for parallel CFD for steady-state flows ends up in requirement for reusable algorithm for solutions partial differential equations and how it can be run on many processors. Trebotich *et*

al. (2008) presented the algorithm to obtained higher performance for the complex geometries for solution elliptical problems and demonstrated how it can be run on 1000 processors.

Another direction in parallel CFD computations is creating mesh-less algorithms. Successful implementation will directly lead to almost embarrassingly parallel CFD. Katz & Jameson (2010), developed a new mesh-less technique scheme based on the well known Taylor series expansions with least squares. The authors discussed difficulties associated with the application of a reusable algorithm for an arbitrary cloud of points. The proposed scheme significantly reduces the storage requirement compared to other mesh-less schemes. They applied this method to the Euler equations and show that this approach agrees with other established methods.

Another question that arises is how to reduce the number of dependencies in iteration solver. The approach was made an example of optimising time steps Arbenz & Obrist (2018) when simulating time-periodic steady states of the Navier-Stokes equations. Two methods were compared, one with a standard time-stepping and another where the time step was recalculating based on periodic boundary conditions in time. The methods are compared concerning accuracy and scalability by solving for a time-periodic Taylor-Green vortex. It has been shown that the second approach converges much faster to simulate equilibrium, reducing the number of dependencies from previous steps lead to the parallel execution of the algorithm.

Our contribution.

In the first part of the thesis, we develop paradigm and methodology for speeding up massive calculations by parallelisation of Navier Stokes equations. We discuss and demonstrate new phenomena which only appear in the parallel world and demonstrate how they can be used in turbulent flow simulation.

We have developed the thread pool class. The object of this class can dynamically add or remove threads to optimise the speed of simulations. This class uses interfaces to allow different numerical schemes to be supplied as parameters. This makes the object of this class reusable. As an example of application, we use this thread pool class into a sequential channel flow solver to execute those regions of the code that are computationally intensive. We demonstrate that

when the number of threads is increased by a factor of two, the speed of the calculation is more than doubled. A significant contribution of this work is that we exploit the benefits of encapsulation which makes develop thread pool class applicable to other CFD applications.

The direct numerical solution of the Navier-Stokes equations is obtained using short time and length scales. As a result, these solutions inevitably contain a large amount of data. In this work, we have developed an approach to rapidly and conveniently analyse the solutions. We demonstrate an approach to deal with the huge amount of data. We have created a highly efficient platform that is intended to be easy to be used by the scientific community to devise and test their sub-grid LES models against the results of DNS. The Johns Hopkins University database of DNS solutions was used for comparison. To help scientists and engineers to evaluate their LES models, we present a comprehensive comparator operator to quantify the accuracy of the models. Furthermore, the method releases researchers from the need to write a comprehensive code because the LES models can be implemented as plugins.

This work has presented an intellectual framework whereby CFD practitioners can readily and quickly examine the accuracy of new models they might wish to propose. The method is based on database technology and includes the following concepts:

- Use and manipulation of heap memory to handle huge volumes of data;
- The implementation of a client-based database engine; and
- The incorporation of efficient fast-Fourier transforms algorithms.

The package is implemented on a high performance computing cluster. An idea permeating the methodology is that a core is defined that contains the ‘know-how’ associated with accessing and manipulating data, and which operates independently of a plugin, this enables users to propose LES models and obtain results almost instantaneously. This research was presented on First Thermal and Fluids Engineering Summer Conference, USA, and subsequentially published in Begell house magazine Grossman (2015).

2 THE STUDY OF CHANNEL FLOW

2.1 INTRODUCTION

The Navier-Stokes equations are derived from the axioms of continuum mechanics and well established constitutive relations. Countless observations and experiments have established their veracity. However, difficulties persist in experimentally quantifying to a high degree of accuracy, even the large-scale features of turbulent flows. The problem is compounded when attempting to measure the smallest length and time scales that may be less than 0.1mm or 1 ms respectively. Although it may seem to contravene the paradigms of the scientific method, scientists and engineers are prepared to accept accurate DNS solutions of the Navier-Stokes equations to establish features of turbulent flows that are not yet amenable to sufficiently accurate experimentation.

The flow between two parallel plates, channel flow, is one of the most straightforward configurations to simulate. The flow may be driven by the relative motion of the two plates, or a pressure gradient may drive it. These features make this flow configuration relatively easy to model, but the results can nonetheless provide us with deep insights into the nature of the turbulent flow.

Lee *et al*. (2015) reported several of the issues that the DNS of channel flow can address. For example, they point out that Smits and Marusic (2013) note that turbulent flows with Re_τ of about 10^3 and higher are technologically significant. This can be appreciated if we consider air at atmospheric temperature and pressure flowing with a velocity of 15 m/s through a straight, circular pipe that has an internal diameter of 10 cm; we find that Re is on the order of 10^5 and $Re_\tau \approx 4\times10^3$.

From an industrial point of view, the preceding example may be considered entirely inconsequential – engineers would almost certainly resort to empirical correlations to calculate the pressure gradient, say, along with the pipe. However, DNS can be a powerful tool to elucidate the nature of turbulent flows. The universal logarithmic law that governs the mean velocity of turbulent flows in the vicinity of walls is described in most fluid dynamics textbooks. In such works, it is generally assumed that von Kármán's constant, κ, is indeed universal

although a range of values is reported in textbooks. Lee et al. (2015) report several studies that indicate κ is not a universal constant but is affected by the geometry of the flow domain. Furthermore, flows with Reynolds numbers, Re_τ, of less than 2,000 appear not to exhibit a Re_τ-independent logarithmic mean velocity profile. This is but one of the many contemporary issues that fluid mechanics are addressing employing computational fluid dynamics.

One of the principal aims of this work is to harness the philosophy and practice of contemporary parallelisation methods, particularly multiprocessing and multithreading. The underlying idea that motivates this work is to identify bottlenecks in serial codes and to devise simple interventions that parallelise the code through thread injection. Gibson (2014) has developed *Channelflow,* which is written as a set of C++ classes and is used to model flow between two parallel plates. *Channelflow* uses spectral discretisation in the three spatial dimensions, and Fourier series are used to discretise the governing equations in the streamwise and spanwise directions, and a Chebychev series is used in the wall-normal direction. As Gibson (2014) points out, the mathematical treatment has been presented by Canuto et al. (1988).

The system researched in this study displayed in Figure 2.1 The stream, spanwise and wall-normal directions are co-linear with the x_1, x_2 and x_3 axes respectively.

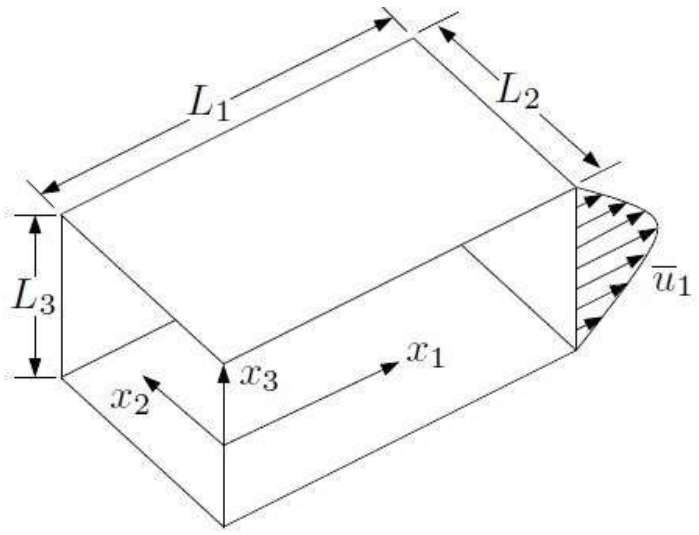


Figure 2.1 The geometry of the system used to study the flow between two parallel plates. The system is semi-infinite in the x_1 and x_2 directions and the fluid velocity at the lower and upper walls is set to zero to conform to the no-slip boundary condition. (Gibson, 2014)

The stream direction denoted as x_1 ,x_2 is the spanwise direction and the x_3 direction is normal to the walls. The extent of the domain is prescribed by L_1, L_2 and L_3 and periodic boundary conditions are imposed on both the streamwise and spanwise directions. The no-slip boundary condition is imposed at the walls.

A fully spectral method used to the discretised Navier-Stokes equations are repeated here for completeness:

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = f_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x}\left(\nu \frac{\partial u_i}{\partial x_j}\right), \tag{2.1}$$

along with the mass conservation and incompressibility constraint

$$\frac{\partial u_i}{\partial x_i} = 0, \tag{2.2}$$

A Fourier representation is used in the wall-parallel direction and Chebyshev expansions in the wall normal directions.

The approximation for the velocities is given by:

$$\overline{u}_i(x_1, x_2, x_3, t) = \sum_{k_1}\sum_{k_2}\sum_{k_3} \widehat{\overline{u}}_i\,(k_1, k_2, k_3, t) e^{2\pi i}\left(\frac{k_1 x_1}{L_1} + \frac{k_2 x_2}{L_2}\right) T_{k_3}(x_3)\widehat{\overline{u}}_i \tag{2.3}$$

where $i = \sqrt{-1}$ and k_j are wave numbers. The spectral velocity is denoted by $\widehat{\overline{u}}_i$.

The Chebyshev polynomial $T_{k_3}(x_3)$ is given by

$$T_{k_3}(x_3) = \cos(k_3 \arccos(x_3)) \tag{2.4}$$

In the wall-parallel direction, the computational nodes are uniformly spaced

$$*\Delta x = {}^{L_1}/_{N_1} \text{ and } \Delta y = {}^{L_2}/_{N_2} \tag{2.5}$$

where N_1 and N_2 are the numbers of computational nodes in $x_1$and $x_2$directions respectively.

The discretization mesh in the direction x_3 is non-uniform and is defined by Chebyshev (Lyle *et al*., 1966)

$$x_{3,j} = cos(j\pi/N_3)\ , \quad 0 \leq \text{j} \leq N_3 - 1 \tag{2.6}$$

Periodic boundary conditions are imposed in the x_1 and x_2 directions. This makes it possible to use the Galerkin method for minimising the residuals. Chebyshev polynomials did not satisfy no-slip conditions, so the tau method is being used. The tau method was discovered by Lanczos (Lanczos et al. 1938) when he worked under Albert Einstein on the theory of relativity. He introduced the use of Chebyshev polynomials for the procedure of finding the solution of linear differential equations with polynomial coefficients.

$$Dy(x) = 0 \tag{2.7}$$

Instead of trying to develop an n^{th} order approximation of equations of the general type given by 2.5 by truncating infinite power series expansions, the method attempts to find an exact polynomial solution to a modified version of this equation. This equation is a perturbed version of equation 2.5 and is obtained by adding to the right-hand side of a polynomial perturbation term. The term is chosen in such a way that it becomes possible to find power series solutions with only a small number of terms.

2.2 THE IMPLEMENTATION OF CHANNEL FLOW

Channelflow is written using an object-oriented development paradigm (Gibson, 2014). It is written as a C++ class library. Instances of how these classes act can be integrated to develop detailed simulations of channel flows. It includes time integration for plane Couette laminar and turbulent flows in the space between two parallel plates, one of which moves relative to the other; pressure-driven channel flow, along with algorithms for computing travelling waves and periodic orbits; and algorithms for computing linear stability of exact solutions. *Channelflow* uses dynamic memory allocation, and each class controls its dynamic memory. The underlying mathematical approach is based on spectral discretisation in the spatial directions, and finite differencing in time. For time stepping it invokes semi-implicit backward differentiation of orders 1-4, two 2nd-order Runge-Kutta schemes, and the classic 2nd-order Crank-Nicolson Adams-Bashforth algorithm. *Channelflow* uses a powerful FFTW library for its Fourier transforms.

As it stands, *Channelflow* is used as an object-oriented paradigm, but it is not ready for the parallelisation. This renders *Channelflow* as a good candidate for implementing parallelisation paradigm. In the following chapters, we will investigate ways to apply parallelisation to *Channelflow* software. Then we will implement a new "thread injection method" which can be applied to a wider range of Computational Flow Dynamics problems.

2.3 PARALLEL DECOMPOSITION

2.3.1 SIMULATION METHOD

Although this approach is quite general, this research will use a *Chanelflow* direct numerical simulator where classes and objects will use to demonstrate this research model.

2.3.2 SPECTRAL METHODS

Spectral methods are a technique where we apply mathematics and physics to solve certain differential equations. The idea behind this is to write the solution as a sum of "basic functions", the coefficients in this sum then choose to satisfy differential equations as well as initial and boundary conditions (Kerr & Kimura, 1998)

Quite often, this is done using Fourier series and requires involving Fast Fourier Transform Technik (Loan, 1992).

The implementation of spectral methods is generally based on using Galerkin (Gottlieb *et al*., 1977) or Tau's methods (Fox *et al*., 1968).

2.3.3 GOVERNING EQUATIONS

The incompressible hydrodynamic turbulence was described by Navier-Stokes equations (1.1, 1.2) in a tensor form, however for parallel decomposition and derivation of spectral methods better suits equations in vector form

$$\partial_t \boldsymbol{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\boldsymbol{\nabla} p + \nu \boldsymbol{\nabla^2} \mathbf{u} \quad (2.8)$$

$$\nabla \cdot \boldsymbol{u} = \mathbf{0} \quad (2.9)$$

where

$\boldsymbol{u}$ - is the velocity vector

p - is the pressure

ν - is the kinematic viscosity

These equations contain:

$(\mathbf{u} \cdot \nabla)\mathbf{u}$ - the term responsible for the advection of momentum and

$\nu\boldsymbol{\nabla^2}\mathbf{u}$ - the term responsible for energy dissipation.

Equation (2.9) represents the incompressibility constraint.

2.3.4 BOUNDARY CONDITIONS AND PRESSURE

To derive an equation for pressure, let's apply a divergence operator to the left and right side of Navier-Stokes equations:

$$\nabla(\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u}) = \boldsymbol{\nabla}(\,\nu\boldsymbol{\nabla^2}\mathbf{u} - \boldsymbol{\nabla}p), \tag{2.10}$$

and substitute incompressibility constraint $\nabla \cdot u = 0$ in (2.3); we will get

$$-\boldsymbol{\nabla^2}p = \boldsymbol{\partial_i \mathbf{u}_j\, \partial_j \mathbf{u}_i} \tag{2.11}$$

This equation is called the Poisson equation for pressure.

To define a complete system of equations we need to specify initial and boundary conditions.

Generally, a pressure condition cannot be used at the boundary where velocities are also specified, because the velocity is derived by a pressure gradient. Usually, the velocity at the boundary is assigned based on zero normal derivatives to the wall $\frac{\boldsymbol{\partial u}}{\boldsymbol{\partial n}} = 0$ as well as zero velocity at the wall $\boldsymbol{u}_w = 0$

2.3.5 SPECTRAL DECOMPOSITION

To derive a spectral decomposition approach, let's assume that:

u(x) is the periodic function in x-direction with period 2π

$$u(x) = \int u(k)e^{ikx}dk \tag{2.12}$$

and where

$$u(k) = \frac{1}{2\pi}\int u(x)e^{-ikx}dx \tag{2.13}$$

By substituting this definition into Navier-Stokes equations, we will obtain the following:

$$\frac{\partial u(x)}{\partial x} = \int u(k)\frac{\partial}{\partial x}e^{ikx}dk = iku(k)e^{ikx}dk \tag{2.14}$$

in 3D space, the above equation will become:

$$\vec{u}(\vec{r}) = \int \vec{u}(\vec{k})e^{i\vec{k}\vec{r}}d^3k \tag{2.15}$$

with the corresponding equation

$$\vec{u}(\vec{k}) = \frac{1}{(2\pi)^3}\int \vec{u}(\vec{r})e^{-i\vec{k}\vec{r}}d^3r \tag{2.16}$$

2.3.6 ENERGY DISSIPATION

We substitute the above definition of u to energy dissipation equation

$$\nu\boldsymbol{\nabla^2 u} \tag{2.17}$$

Also, because we transfer $\partial_x^2 + \partial_y^2 + \partial_z^2$ in Fourier space

$$\partial_x^2 + \partial_y^2 + \partial_z^2 \rightarrow (ik_x)^2 + (ik_y)^2 + (ik_z)^2 = -|\mathbf{k}|^2$$

we obtain the following formula for energy dissipation:

$$\nu\boldsymbol{\nabla^2} u(\boldsymbol{r}) = \nu\int -|\boldsymbol{k}|^2\boldsymbol{u}(\boldsymbol{k})\boldsymbol{d}^3\,\boldsymbol{k} \tag{2.18}$$

This assumes that we know the nonlinear term:

$$F(r) = (u(r) \cdot \nabla)\mathrm{u(r)} \tag{2.19}$$

and the corresponding term in Fourier space F(k).

Then the diffusion equation can be solved in time exactly and efficiently in Fourier space, and the forced Navier-Stokes equations in physical space are:

$$\dot{\boldsymbol{u}}(\boldsymbol{r}) + \boldsymbol{F}(r) = \frac{\partial}{\partial t}\boldsymbol{u} + \boldsymbol{F}(\boldsymbol{r}) = -\nabla p + \nu \nabla^2 \boldsymbol{u} \tag{2.20}$$

and the corresponding equation in Fourier space is

$$\dot{\boldsymbol{u}}(\boldsymbol{k}) + \boldsymbol{F}(k) = -i\boldsymbol{k}P(k) - \nu|\boldsymbol{k}|^2\boldsymbol{u}(\boldsymbol{k}) \tag{2.21}$$

Including incompressibility $\boldsymbol{k} \cdot \boldsymbol{u} = 0$ we will get the following,

where the pressure equation is reduced to:

$$|\boldsymbol{k}|^2 P(k) = i\boldsymbol{k} \cdot \mathbf{F}(\mathrm{k}) \tag{2.22}$$

Instead of having to invert the pressure equation in physical space, all terms are linear in the Navier Stokes equations in Fourier space

$$\dot{u}_i\,(k) + \left(\delta_{ij} - \frac{k_i k_j}{k^2}\right) F_j(k) = -\nu|k|^2 u_{\boldsymbol{i}}(k) \tag{2.23}$$

2.3.7 FOURIER TRANSFORMATION OF THE NON-LINEAR TERM

Returning to equation (2.12) in Fourier space,

$$F_i(k) = \int \mathrm{d}^3 r e^{-ik\cdot r}\left(\int \mathrm{d}^3 q e^{-iq\cdot r} u_j\ (q)\right)\left(\int \mathrm{d}^3 p e^{-ip\cdot r} u_j\ (p)\right) \tag{2.24}$$

and

$$\int \mathrm{d}^3 r e^{-ik-p-q)\cdot r} = \delta(-k+p+q) \tag{2.25}$$

$$\partial_j e^{-ip\cdot r} = ipe^{ip\cdot r} \tag{2.26}$$

the non-linear function F(k) in Fourier space is:

$$F(k) = \sum_{k=p+q} i\big(\boldsymbol{p}\cdot\boldsymbol{u}(\boldsymbol{q})\big)\boldsymbol{u}(\boldsymbol{p}) \tag{2.27}$$

or

$$F_i(k) = \sum_{k=p+q} i\left(p_j\cdot u_i(p)\right)u_j(q) \tag{2.28}$$

Let us calculate how computationally expensive is this term.

For the grid with n nodes, mesh size is about

n^3 and roughly it requires n^3 wavenumber operations. To calculate non-linear terms for each k we have to do another n^3 wavenumber sum.

The total number of operations is $n^3 \times n^3 = n^6$

This makes the nonlinear convection term in the Navier-Stokes equations the most computationally expensive, and we will show how we can apply a parallelisation object-oriented approach to eliminate this obstacle.

Let us continue to analyse of computational expense for full Fourier transformed Navier-Stokes equations:

$$\dot{u}_i\ (k) + \left(\delta_{ij} - \frac{k_i k_j}{k^2}\right)\sum_{k=p+q} i\left(p_j\cdot u_i(p)\right)u_j(q) = -\nu|k|^2 u_i(k) \tag{2.29}$$

In the above equation, the linear terms are evaluated efficiently in Fourier space. However, the non-linear term is expensive. Furthermore, linear terms involving the evaluation of Laplacians are also expensive to compute. In this research, we have adopted the strategy of expressing the non-linear terms as a finite difference approximation, i.e. Equation 2.30 is simply a numerical representation of a finite element derivative, and it forms a component of the derivation.

$$u\partial_x u \rightarrow u(x_i)\frac{u(x_{i+1}) - u(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{2.30}$$

2.3.8 SIMPLIFY SPECTRAL METHODS AND ALIASING PROBLEMS

To summarise, all the above spectral methods are based on the following steps:

- Calculate derivatives exactly in Fourier space;
- Revert velocities and derivatives to physical space;
- Calculate non-linear terms;
- Transform non-linear terms to Fourier space; and
- Solve Navier-Stokes equations.

However, this approach creates aliasing errors. Backward and forward Fourier transformation gives in additional terms responsible for aliasing problems. For example, *n*-grid points produce n real numbers u(x_i), $i = 1,2, \ldots n$. Then *n*-Fourier coefficients have *n* complex numbers, which are 2*n* real numbers. So, when multiplying the non-linear terms, we will have the following:

$$u\left(k = \frac{n}{3} + 1\right)u\left(k = \frac{n}{3} + 1\right) = NL\left(k = \frac{2n}{3} + 1\right) = NL\left(k = -\frac{n}{3} + 1\right) = NL\left(\frac{n}{3} - 1\right) \tag{2.31}$$

Two high wavenumbers which should create a still higher wavenumber create a lower wavenumber. There are several attempts to leverage accuracy and computational costs. To overcome this problem, one of the approaches is using truncation all |k| > n/3. This is called the 2/3rds number because we are keeping 2/3 of the wave numbers

1/3 for $-n/3 < k < 0$

and

1/3 for $0 < k < n/3$

2.3.9 This work approach

This work proposes that instead of simplifying and using a truncated Fourier transform described above, it uses an Object-Oriented Approach to apply parallelisation to massive but accurate calculations on multiple computers. This will reduce simulation times due to its ability to do many calculations in parallel, and this is a key motivation of this research. This approach is based on the fact that most time-consuming calculation of the non-linear term of the Navier–Stokes equations

3 GOING PARALLEL

3.1 INTRODUCTION

The parallel world of computing is quite different from the serial world. Some aspects of this will be discussed below in more detail.

Parallelisation is one of the modern and powerful ways in computational fluid dynamics to solve Navier-Stokes equations. However, this presents the challenge of solving the equations simultaneously by making use of many building blocks or threads. When a computer code runs on some blocks, a new paradigm is considered because running in parallel generates scenarios which do not exist in the sequential world of computing and this requires the use of new terminology and language which we shall now briefly discuss.

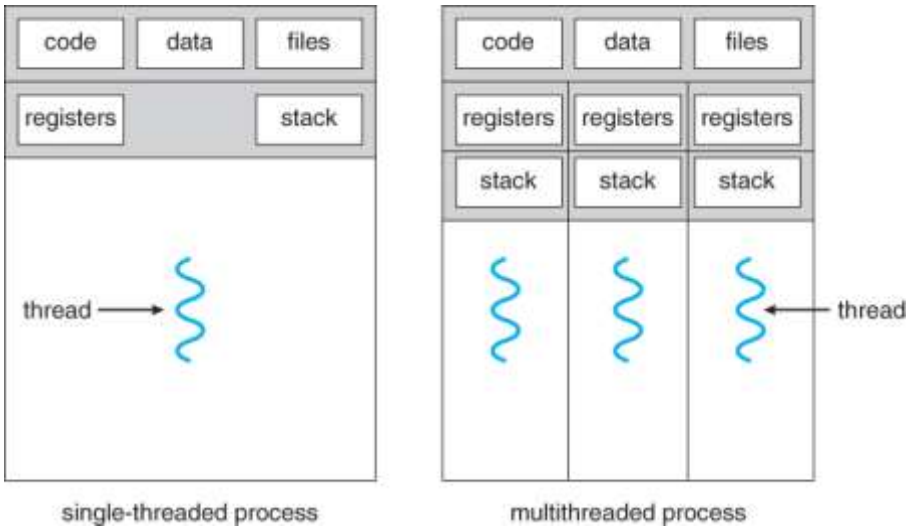


Figure 3.1 A thread can be considered to be a stream which carries a list of computer instructions that are to be executed independently (Silberschatz et al., 2012)

In the sequential world, only one thread exists which is known as the main thread. In parallel systems, there are many threads within the same process, and each of these threads contains its data, a list of executable instructions, a stack and the set of registers.

Thread function

To create a new thread, we call pthread_create() or a similar function depending on the operating environment and computer language. After a thread is created the first thing it starts doing is to execute start routine () – called thread function – passed as the sole argument of the thread creation function. Because threads can coexist, we have computer logic written in each thread function run in parallel. This architecture is a door from sequentially executed computer instructions to the world of parallel computing.

In Computational Fluid Dynamics, we consider that the flow consists of several independent streams, and the multithreading computer paradigm is much closer to the physics of what we are trying to simulate. As to thread function, in CFD, for example, this feature may hold a list of instructions on how to navigate along with the computational domain or sub-domain and compute some of the properties like, for instance, velocities or pressure distribution.

If computed variables are dependent on global space or dependent on results of calculations of another thread, then thread synchronisations may be required.

Parallel Overheads

Parallel computing environments offer the possibility of speeding up execution, but they introduce certain execution time-consuming overheads. These expenses are associated with the amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overheads can include factors such as (Shen et al., 2004):

- Time to start-up;
- Synchronisations;
- Data communications; and
- Time to terminate the thread.

Thread-safe

In computational fluid dynamics, often the value of the flow field velocity in the grid depends on the value of its neighbours. To get the right result, an order of calculations becomes imperative. This leads us to thread-safe calculations.

To be thread-safe, the program should protect shared data; for example, one thread trying to read the value of the shared data when at the same time, another thread wants to modify it. The multiple threads running in an application may create potential issues regarding safe access to resources. This may create a scenario where a program runs in unintended ways. For example, one thread might override other changes or put the application into an unknown and potentially invalid state. In a real case scenario, the corrupted resource might cause obvious problems like performance issues or crashes. In a worst-case scenario, however, corruption may cause serious errors that do not manifest themselves until much later. This kind of problem might require a significant overhaul in the software development process.

The scenario described above can be a serious issue in the numerical simulation of turbulent flow. As already mentioned, the scale of the physical tasks can be considered as large data with up to petabyte related variables. Access and calculation related variables by number simultaneously running threads can be potentially very harmful, as the value of the variable can be corrupted based on the random order of accessing it. If this happens, it is a very challenging task to debug a problem of such scale. The best protection when it comes to thread safety is good design. Avoiding shared resources and minimising the interactions between threads makes it less likely that those threads will interfere with each other. However, an entirely interference-free design is not always possible. In cases where threads must interact, synchronisation tools are required to ensure that when they interact, they do so safely. Developing trade-safe code requires the needs of the software primitive, which will be able to take control over random access to the shared variables and be able to synchronise it. One of the ways to do that is to use Mutexes, which are the abstraction that can also be used for synchronisation.

Mutex - Suppose we have the shared variable which can be accessed and changed by two threads. For example, in our work on parallelising the solution of the Navier-Stokes equations, there may be a possibility that two threads simultaneously access a velocity flow field variable. We must ensure that our algorithm does not inadvertently overwrite this variable. This is achieved by invoking a mutex – a mutually exclusive lock that acts as a protective wall around a resource. We can view the mutex as a traffic light which works like a semaphore that grants access to only one thread at a time. A mutex is an object of the class responsible for synchronisations in between threads. If we did not have a mutex, then the value of this variable would be random depending on which thread accesses this variable first.

In computational fluid dynamics, we may need mutex primitive, for example, in the following scenario. Say we have one thread which is solving differential equations and produces the array of velocity vectors. Another thread will need to read this array and then draw on the screen an image of this velocity field. Using mutexes allows this to synchronise in such a way that as soon as the first thread finishes its calculation, the second thread starts to draw.

To fulfil its duty, a Mutex class usually has the following methods:
Lock() – If any thread does not currently lock the mutex, the calling thread locks it until the same thread unlock is called.
Unlock() - Unlocks the mutex and releases ownership of it.
If the mutex is currently locked by the same thread calling this function, it produces a deadlock (with undefined behaviour).
Dead-lock – does not exist in the sequential world but is a common problem in multithreaded or multiprocessing systems, parallel computing or distributed systems where process synchronisation uses locks.
This mechanism by which the dead-lock operates is illustrated in Figure 3.2. It can be observed that Thread1 requests resource A and get it. Then Thread2 requests resource B and gets it. Then Thread1 requests resource B and starts waiting for resource B, which is locked by Thread2. Then Thread2 requests resource A which is blocked by Thread1. In the above scenario, Thread1 and Thread2 end up in an infinitive waiting stage called deadlock. Another example of deadlock can be a scenario where the mutex is currently locked by the same thread called lock function and produces a deadlock with undefined behaviour.

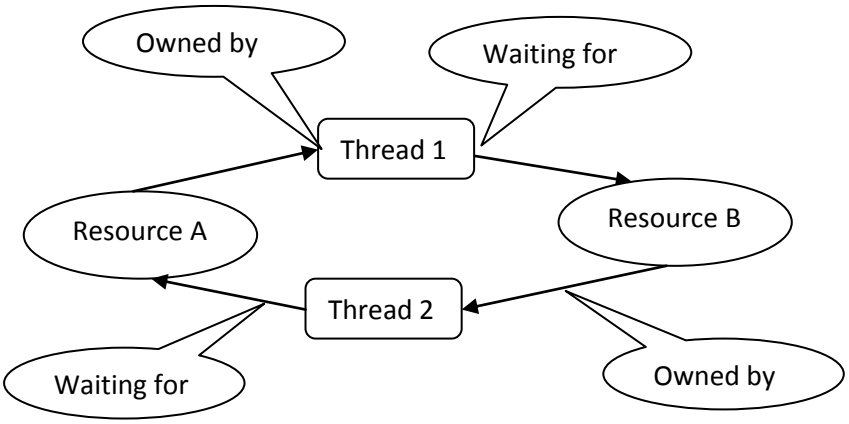


Figure 3.2 Dead-lock schematic example

Before describing our development, we will be narrating what building blocks for parallel simulation are available.

3.1.1 PROCESSES

Here we recount the steps required to generate a computer program. First, the computer code needs to be created. Computer or source code is a text used to write instructions which a computer can interpret, and is done by using computer languages. As with any language, computer languages are split into two components – syntax and semantics. The next step is to compile the source code, which will validate the syntax and produce a positive result that will generate objects. Those objects are important binary files which a computer understands. The last step is when the object code is passed to the linker. When this is done, an executable program is created. However, it still needs to be loaded into memory.

This process is an instance of an executable program. If, for example, four people are running the same program, there will be four processes running simultaneously, not one. We may have more

than one process running with only one person starting the program. This is due to the ability of the process to clone some concurrently running processes.
Each running process is identified by a unique Process Identification Number (PID). This number is assigned when the process is created.
If we run the command on UNIX terminal shell
> ps –ef
operational system (OS) will respond with a list of all running processes.
OS is the time-sharing system, and each process allocates a time slice in its turn to run on CPU. So even when we have a little job and it stays in a queue behind the large job it is executed without delay.
In applications in computational fluid dynamics we target that, some processes will solve one task and particularly Navier-Stokes equations.
To be able to build such systems, processes should not run independently but rather together and be able to communicate with each other. We need to be able to bypass the OS time slice mechanism and make processes synchronise so that they execute and even stop and wait until other processes finish their steps.
We will now discuss what options we have and how this done.

3.1.2 FOREGROUND AND BACKGROUND PROCESSES

If we do not want to wait for a process to finish, we can start the process in the background. Multi-tasking operational systems allow multiple processes to be run in the background and foreground. Adding ampersand at the and of command lets OS know that the process is executed in the background – $ command &. After starting a process in the background, OS returns the shell to the user so that it can continue.
Within these memory limits, multiple processes can be started, running concurrently in this way. A foreground process is different from a background process in two ways. Firstly, foreground processes may show the user an interface through which he/she can interact with the program and secondly, the user must wait for one foreground process to be completed before running another one.

We can use this feature to separate simulation processes of solutions of Navier-Stokes equations on several processes. For example, one process can be responsible for visualisation, another for solving the system of the equations and another one can manage these two processes and synchronise those using interprocess messages.

3.2 INTERRUPTS AND SIGNALS

The paradigm we are trying to build will solve Navier-Stokes equations and do the numerical simulation based on a number independent CPUs which require being able to communicate with different processes and involve a different part of calculations based on physics. The signal is a trigger that is used to notify the processes about the occurrence of a particular event. Signals are often used as a mechanism for inter-process communications. (IPC) interrupts are conceptually similar to signals and can be viewed as triggers to communicate between the CPU and OS kernel, rather than as signals responsible for communication between the OS kernel and OS processes.

To implement multitasking, we can use a hardware interrupt. In the next chapter, we demonstrate how interrupts are used in the implementation of the thread pool class, which we develop for thread injection as a method of speeding up Navier-Stokes simulations.

3.3 SOCKETS

When we are targeting the task of solving Navier-Stokes equations on many simultaneously running computers, we need to have a mechanism to exchange data between them.

And this mechanism is based on using sockets. A socket is an object which allows us to send and receive data in between processes. Sockets are a fundamental component of inter-process and intersystem communication. They provide point-to-point, two-way communication between processes. The socket is an abstraction that provides a set of protocols and allows them to exchange data.

3.4 EVENT DRIVEN COMMUNICATIONS

Moving in the direction of parallel development reflects the need to change a computing paradigm. Instead of acts like in Procedural Programming, parallel development needs the software's ability to react. In general, event-driven communication is programming where the primary activity is a reaction to semantically significant signals (events).
In sequential programming, the flow of calculations is followed by the subsequent run of the instruction. As the thesis aims to study parallel development and turbulence flow, the very logic is to have a software paradigm which more closely reflects the physic of our phenomena. So, we can define an event as to where the flow of calculations determines a reaction to messages received from other processes or threads.

3.5 MULTI-THREADING

Multithreading is a particular form of multitasking. We may divide multitasking into two main types: process-based and thread-based.
Process-based multitasking is about the concurrent execution of programs, while thread-based multitasking deals with simultaneous execution of pieces of the same program.
Thread-based multitasking consists of some parts that can run concurrently, and each of these pieces is called a thread. Each thread is defined by its logic in the path of execution.
We can see multithreading as multiple execution agents working on solving common problems operating simultaneously.
Using multi-threading may create many benefits for computational flow dynamics (Drysdale, 2007), such as:

- **Responsiveness** – for example, one thread can provide a quicker response than another when doing massive calculations;
- **Resource sharing** – threads share common code, data, and other resources that create an environment for multiple tasks to be performed simultaneously in a single address space;
- **Performance** – switches between threads and context are much faster than performing the same tasks for processes; and

- **Scalability** – a single-threaded process can only run on one CPU, regardless of how many CPU are available. However, multithreaded processes can benefit from multicore and multiprocessor architecture.

In the next chapter, we will discuss a ThreadPool class which we develop to use multithreading to speed up solutions with Navier –Stokes equations.

3.6 MULTIPROCESSORS AND MULTI-CORE

A Central Processor Unit (CPU) contains many discrete parts such as instructions decoders, memory caches, and executions units. Multiprocessor systems have more than one CPU allowing them to work in parallel. Multi-Core CPUs has multiple execution cores inside of one CPU. Multiple cores can also work in parallel in separate operations. This architecture is very flexible and allows a very complex system for parallel operations to be established. For example, we can create a multiprocessor multicore multithreaded system (Darlington et al., 1996).
In a modern computer, architecture is quite common to produce chips with multiple cores on a single chip. A multi-threaded application running on a single-core chip would have to intersperse the threads, as shown in Figure 3.1. However, on a multi-core chip, the threads could be spread across the available cores allowing real parallel processing, as shown in Figure 3.3.

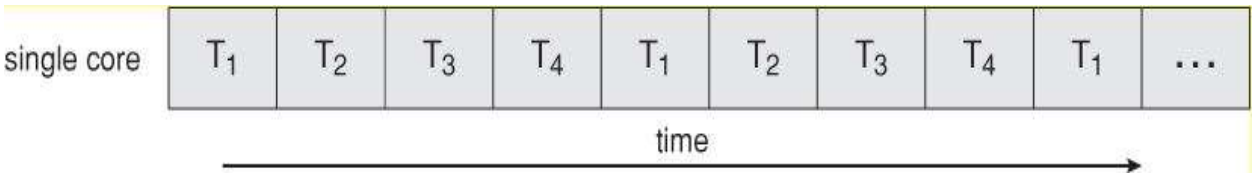


Figure 3.3 Concurrent execution on a single-core system

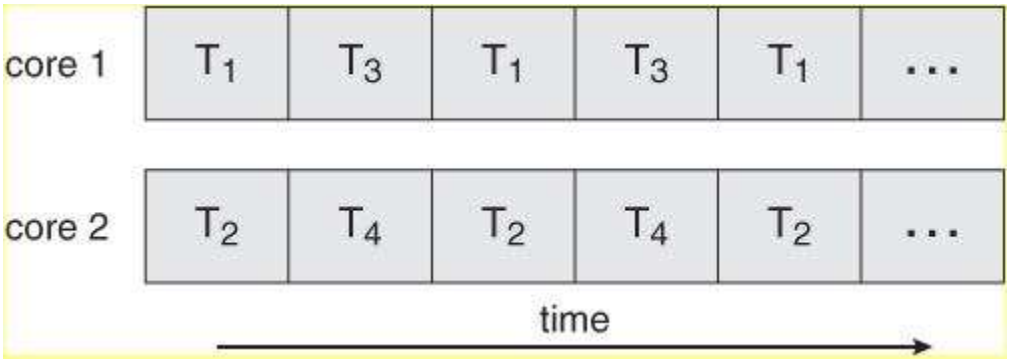


Figure 3.4 - Parallel execution on a multi-core system

In Figure 3.3 and Figure 3.4 T_1…….T_n represent simultaneously running threads, where each one carries instructions for OS to follow. This example demonstrates how threads run on single or multiple core systems.

On the one hand, hardware development boosted demand for new software algorithms for multi-core chips. On the other hand, multithreading software became more and more pervasive. Applications started using thousands instead of tens threads, and this boosts new demand for new hardware where CPU's can support more threads per core. This cycle of rapid growth creates enormous opportunities for computational flow dynamic modelling, where bigger and more realistic simulations are performed (Lavington et al., 1998).

3.7 MPI

As already discussed, processes communicate using messages and events which are a trigger on such messages. It is very logical to build a stand-alone platform which will be responsible for this messaging mechanism. Message Passing Interface (MPI) is a stand-alone abstraction that allows multiple programs to communicate using queues and non-OS managed channels. We can see MPI as clear inter-process communication (IPC).

In a parallel environment, multiple execution agents work concurrently to solve a common task. This agent is not necessarily even located on the same physical server. All the above creates the need for another level of communications and this where MPI becomes useful.

Figure 3.5 MPI allows the creation of a message interface in between two processes to send a message, size, type, source, destination, tag, communicator, status etc…

3.8 OpenMP

Another way for processes to communicate is to use shared memory. OpenMP is a system specification for a set of directives, library routines, and environment variables that can be used to specify shared-memory parallelism.

It uses a portable and scalable interface for developing parallel applications running from a desktop computer to a supercomputer.

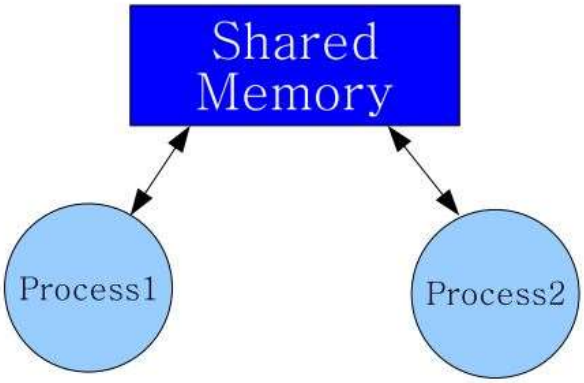


Figure 3.6 OpenMP shared memory management. Processes can communicate by accessing the memory which can be shared in between different processes.

3.9 GPU AND CPU DEVELOPMENT

A graphics processor unit (GPU) was originally designed to speed up computer graphics. The recent discovery that GPU can be used not only for accelerating graphics applications but also for massive computer calculations has taken the scientific world by storm. This all happened due to the dramatic increase in the speed of calculations on GPUs. Even the fact that a single core in a GPU unit is relatively slow compared with CPU provides a significantly bigger number of simple, data-parallel and deeply multithreaded cores and very high memory bandwidths.

GPU architecture involves more software developers as it has a potential for dramatically increasing the speed of applications, and especially computational flow dynamics as it is a very time-consuming process (Kruger et al., 2005).

3.10 OBJECT-ORIENTED LANGUAGES AND DESIGN

So far, we have seen that the history of operation system evolution has been moving towards reusable components. These components more or less encapsulated and were intended to serve various requests or tasks. Software languages started to follow this pattern. Originally computer

languages were designed to perform functional and sequential operations, but recently they became object-oriented. Here, it is probably necessary to define the object.

An object is an abstraction which reflects typical properties of the real-world system which we are going to simulate and about which we want to store information. An object could be a differential equation, flow field, temperature sensor, etc.
One example of an object would be a shape used in CFD for mesh generation development.

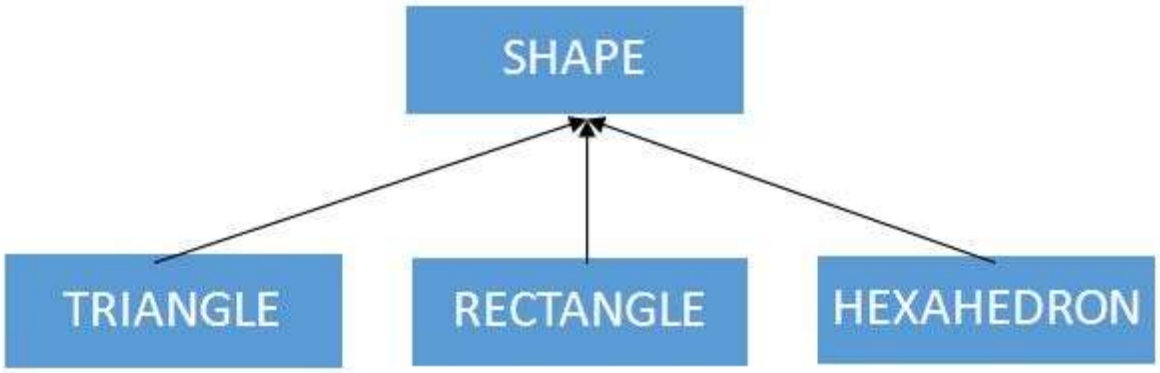


Figure 3.7 Example of the object – Shape

There are many different objects in the real world, such as dog, desk, television. They always exist, but only recently, the concept of objects started to be encapsulated in computer languages.

When we consider real-world objects, we may see that they share two general characteristics: state and behaviour. Identifying the state and behaviour of real-world objects is a first step in creating an architecture for object-oriented development. Real-world objects vary in complexity, as some objects may contain other objects.

These real-world observations all translate into the world of object-oriented programming.
In software development objects are similar to real-world objects: software objects also consist of state and behaviour. A computer object stores its state in variables and exposes its behaviour

through methods. Methods deal with an object's internal data state and work as the mechanism for object-to-object communication.

Bundling computer code into individual software objects provides some benefits and makes applications more modular. The source code for an object is written and maintained independently of the system of equipment for other parts. When the process of object instantiating finishes, it may pass around and start to reuse.

This approach is a very convenient model to apply to CFD and parallelisation paradigm. As we can see, we can use the idea of objects to create independent data types such as vertex, points, and streams. Then building a CFD model becomes more closely aligned to physics.

Information-hiding: by interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world. Information hiding is another very useful paradigm which has a direct connection to turbulent flow study; for example, such abstractions like eddies in LES can be viewed as information hiding entities to produce macro properties for the turbulent flow, like pressure or temperature distribution. Object-oriented programming has become the most widely used approach to software development.

3.10.1 OBJECTS IN PROGRAMMING

Object-oriented programming (OOP) is a methodology for problem-solving where all computations are performed by using objects. The code in object-oriented programming is organised around objects. Once objects are defined, they can interact with each other to make something happen; for example, we need an application where a person gets into a car and drives it. To do so, we will define the required objects, such as a person and car. That includes methods: a person knows how to drive a car and a car knows what it is like to be driven. Once objects instantiate, they can be brought together so the person can get into the car and drive.

In this research, we create and use some stand-alone objects. In particular, each algorithm now is an object. We have objects responsible for memory allocation when we are going in significant data areas etc. The detailed list of developed classes that describe this job objects is shown in chapters 4 and 5.

3.10.2 CLASSES AND OBJECTS

A class is a design for the ultimate object. The class is considered as a concept and object is the implementation of this idea. Classes are very useful in programming; for example, if we need to model 1000 people rather than one person. Instead of describing each one in detail, we may create 1000 objects of type 'person' and then allocate properties to them like name, address, etc. And all this can be done by using only one class – person.

Similar to computational flow dynamics, the objects oriented approach is very useful to apply. For instance, consider that we need to describe and model computational grid or mesh, and the mesh can consist of 2D objects like triangles or quadrilateral, or 3D objects like tetrahedral shapes. An object-oriented approach to doing that will be to define the class shape. Then set a class triangle, class Quadrilateral, and class Tetrahedral. All these classes derive from one based class – Shape. So, to describe hundreds of thousands of mesh cells we only need four Classes, and then we create our hundred thousand objects which are ready to communicate as one object called mesh. The benefit of this approach is that it directly links to parallel and distributed computer systems. Here we can see two different types of Classes.

The first type is the shape class. There is no object of this Class that we anticipate need to be created. Classes like that are called Abstract Classes.

The second type is a Class that represents geometrical shapes – Triangle, Quadrilateral, and Tetrahedral. These types signify more complex objects as they all have another object inside – shape – as they are derived from it. The class shape then becomes an interface to manipulate objects of the different types that derived from it.

3.10.3 METHODS AND FUNCTIONS

The method can be seen as an action that an object can perform, and that defines the behaviour of the objects which are created from the class.

A function is a combination of instructions that are merged to achieve something, and typically requires some input (called arguments) and returns some results.

How is the function different from the method? A function is independent, whereas the method always belongs to the class which represents the object. The function can be used anywhere in

the code and don't need to have an object to use it. However, methods always stay with their object.

All the above objects make reusable building blocks. To summarise, the object is an abstraction of something that exists in the real world or in our minds which belongs to the system we want to model and about which we want to store information (Shelly et al., 2008).

The development of the object-oriented paradigm is now briefly outlined. It stemmed from the initial ideas of a new programming approach, while the design and analysis methods came much later (Dahl et al., 2004). The first object-oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center. Then in 1972, Alan Kay and his colleagues at Xerox PARK created the first pure object-oriented programming language (OOPL), Smalltalk, for programming in the first personal computer Dynabook (Kay,1972).

Grady Booch G.,(1982) published a paper titled Object-Oriented Design that initially presented a model for the programming language - Ada. And in the following editions, he extended his ideas to a complete object-oriented paradigm. Schlienger, F.et *al*.,(1994) presented ideas to object-oriented methods. The other substantial innovations were Object Modelling Techniques (OMT) by Rumbaugh, J. et *al*.,(1990) and Object-Oriented Software Engineering (OOSE) by Jacobson, I., (1992).

3.11 OBJECT-ORIENTED ANALYSIS

Object-Oriented Analysis (OOA) is the methodology of identifying independent software engineering requirements and developing specifications regarding object models.

The object-oriented analysis differs from other forms of analysis as it requires and is organised around objects; they are modelled after real-world objects and system built based on the interaction between objects. In traditional analysis, methodologies, functions, and data are considered separately.

The object-oriented analysis (OOA) started from identifying objects; then objects needed to be organised by creating a model diagram following the definition of the internals of the objects, or object attributes. Part of OOA is defining the behaviour of the objects; i.e., object actions and describing how the objects interact.

The aforementioned described steps are utilised in this research paper. When building a paradigm for numerical simulation Navier-Stokes equations in this research, I am trying to identify stand-alone parts of the algorithm, then convert them to objects, identifying properties of those objects and seeing which of them can be used in parallel. Then identify bottlenecks at runtime and use created objects to take control of existing sequential calculations to split them into some independently running threads

3.12 OBJECT-ORIENTED DESIGN

Object-Oriented Design (OOD) involves the implementation of the conceptual model produced during object-oriented analysis. The implementation steps usually include:

- If necessary, data restructuring of the Class;
- Developing source for methods; i.e., internal data structures and algorithms; and
- Developing source code for controls and associations.

3.13 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a paradigm based on using objects. The main aim here is to incorporate the advantages of modularity and reusability. Objects, which are instances of classes, are used to interact with one another to fulfil application design and computer programs. The important features of object-oriented programming are:

- Approaching program design from the bottom to the top;
- Organising an application around objects, and grouping them in classes;
- Developing data and methods to operate on an object's data;
- Designing interaction between objects through its methods; and
- Designing reusability of the object by creating new classes and adding new features to existing classes.

3.14 OBJECT-ORIENTED PRINCIPLES

3.14.1 ENCAPSULATION

From an object-oriented point of view, the object is a fundamental building block. We start using object concepts by dividing development and design into two parts.

The first part is logic and functionality inside the object. The second part is public interfaces – what object is used to communicate with other regions of the application.

Development and know-how inside the object logic do not always need to be visible to the user. In other words, encapsulation is about hiding complexity. In the real world, objects quite often hide their information and how they work; we don't need to know the internal details of the object.

When we create an object in an object-oriented language, the complexity of the inner workings of the object can be hidden.

For example, a computational flow analyst can get a solution to his task by using objects in a simulation package without knowing the mathematical methods encapsulated inside the objects that this simulation package uses.

Information hiding is a key in object-oriented design as it allows anyone to use the object and reuse it if needed. Another reason for hiding complexity is to manage changes.

Any big system at present is almost always going to be in a new development cycle where new features and new functionality are added, and this may require making some changes inside one particular object. However, overall it does not affect the whole system. When changes inside the object are completed, they will automatically be reused by rest of the system.

A detailed example in CFD is, for instance, that almost always any numerical simulation requires a system for solving equations. Creating a solver object will separate use of this purpose and the object implementation. When or if the solver object will need to have an upgrade for whatever reason, it will not affect users, and the system will continue running.

The encapsulation of the object is controlled by a public and private keyword to grant access or remove it from the different parts of the object, where private and public methods of the object became handy.

3.14.2 INHERITANCE

Inheritance – the ability to derive something specific from something generic – can be encountered in everyday life. In Object-Oriented inheritance, it enables new objects to take on properties of existing objects. There is always an excellent way to reuse existing functionality rather than create the same thing again and again, and an essential feature of the Object-Oriented approach is reusability. Reusing the properties of the objects lets us not only save time and money but also makes the application more reliable.
Inheritance allows a software developer to write clearer code as the complexity is reduced by reusing similar properties and sharing code between derived objects.

3.14.3 POLYMORPHISM

The word Polymorphism comes from Greek and means “having multiple forms”.
In object-oriented programming, this is the characteristic of being able to assign a different meaning or usage to various entities such as variables, functions or objects that have multiple forms; in other words, polymorphism describes a pattern in object-oriented programming in which classes have different functionality while sharing a common interface. Polymorphism can be of two types – static and dynamic.
In dynamic polymorphism, the response to the message is decided at runtime while in static polymorphism it is decided on compilation time.

3.15 OBJECTS IN CFD

Development in parallel and distributed systems has resulted in dramatically increased computational power and efficiency. From the implementation point of view, modern

programming languages offer potent tools for flexibility, such as the inheritance of object-oriented programming.

In our research, we develop computer code which is based on highly parallel principals for solving Navier-Stokes equations for incompressible turbulent flows. Also, we design and implement modular mathematical abstractions objects which are reusable in many simulation applications.

4 OBJECT-ORIENTED DEVELOPMENT AND PARALLELIZATION OF THE NONLINEAR CONVECTION TERM

4.1 INTRODUCTION

Here we are back to developing an Object-Oriented Approach to compute a $\boldsymbol{u\nabla u}$ - non-linear term in Navier-Stokes equations which is responsible for the transfer of kinetic energy in the turbulent flow. There are some numerical algorithms to calculate this term:

The convection form $$\boldsymbol{u \cdot \nabla u} \quad (4.1)$$

The divergence form $$\boldsymbol{\nabla \cdot (uu)} \quad (4.2)$$

The skew-symmetric form $$\frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{u \cdot \nabla u} + \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{\nabla(u \cdot u)} \quad (4.3)$$

The rotational form $$\boldsymbol{(\nabla \times u) \times u} + \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{\nabla(u \cdot u)} \quad (4.4)$$

These expressions are numerically equivalent but have different calculation costs. When discretised the rotational form is less expensive to compute, but it introduces some errors in the high spatial frequencies, which can be reduced by applying a de-aliased transformation (Mitchell et al., 1988).

The calculation cost of the skew-symmetrical method lies between the convection and divergence forms and is free from such errors. However, it is twice as expensive to calculate. This averaging is simulated by alternating between the convection and divergence forms on successive time steps. Such an approach produces excellent results as in practice, the skew-symmetric method is almost as fast as the rotational method.

Krist and Zang (1987) recommend using the skew-symmetric or alternating forms with aliased transforms or the rotational form with idealised transforms. The *Channelflow* application implements the rotational, convection, divergence, skew-symmetric, and alternating forms. The computational algorithms of each of the methods assume a common form; i.e.

```
for (int i=0; i<3; ++i)
        for (int j=0; j<3; ++j) {
        int ij = i3j(i,j);
                for (int ny=0; ny<Ny; ++ny)
                for (int nx=0; nx<Nx; ++nx)
                for (int nz=0; nz<Nz; ++nz)
                        f(nx,ny,nz,i) += ……..
```

It is these five nested loops that open the door to speed up calculations using threads. To achieve this, we wish to introduce thread injection at a localised point in the existing *Channelflow* serial code.

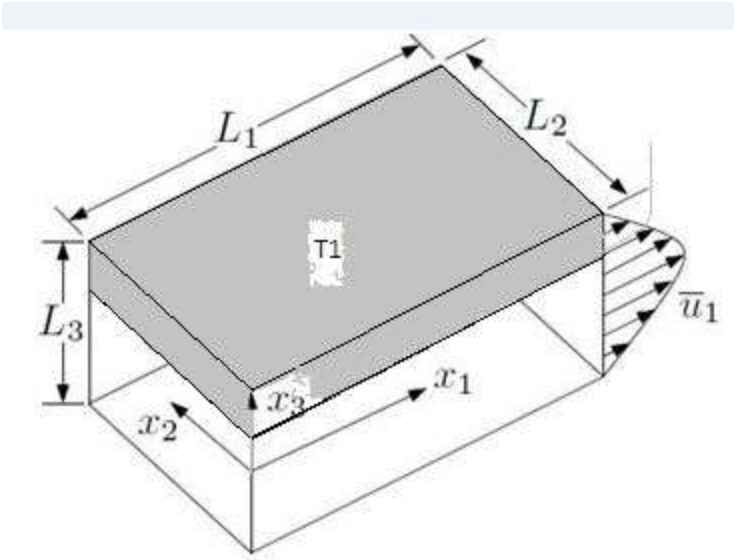


Figure 4.1 Discretization of the domain using a scheme that is amenable to parallelisation

The primary area is divided into slices as shown in Figure 4.1 Instead of traversing the entire domain, and each subdomain is traversed concurrently by individual threads. Thread function will receive one of the above computational methods through a parameter. When a thread completes its task, it should wait until all other threads have also completed their tasks. When all threads have finished their tasks, they have to return to the main thread.

As the thread creation process takes some time, the threads should be created only once at the beginning of simulations. When a thread has completed its task, it should just stay waiting for another job to pass into the thread function as a parameter. To be able to do such a thing, our design should consider a way of talking to threads without stopping and starting them. Firstly, we develop a stand-alone Thread Pool class. The object of this class will be to hold all our threads in preparation for them to execute their assigned computations. Creation of the threads has overhead, so we want the pool to be created only once and destroyed only when the simulation finishes.

The pool will use a hardware interrupt as a signal to the process to communicate with the main thread and notify it with instructions.

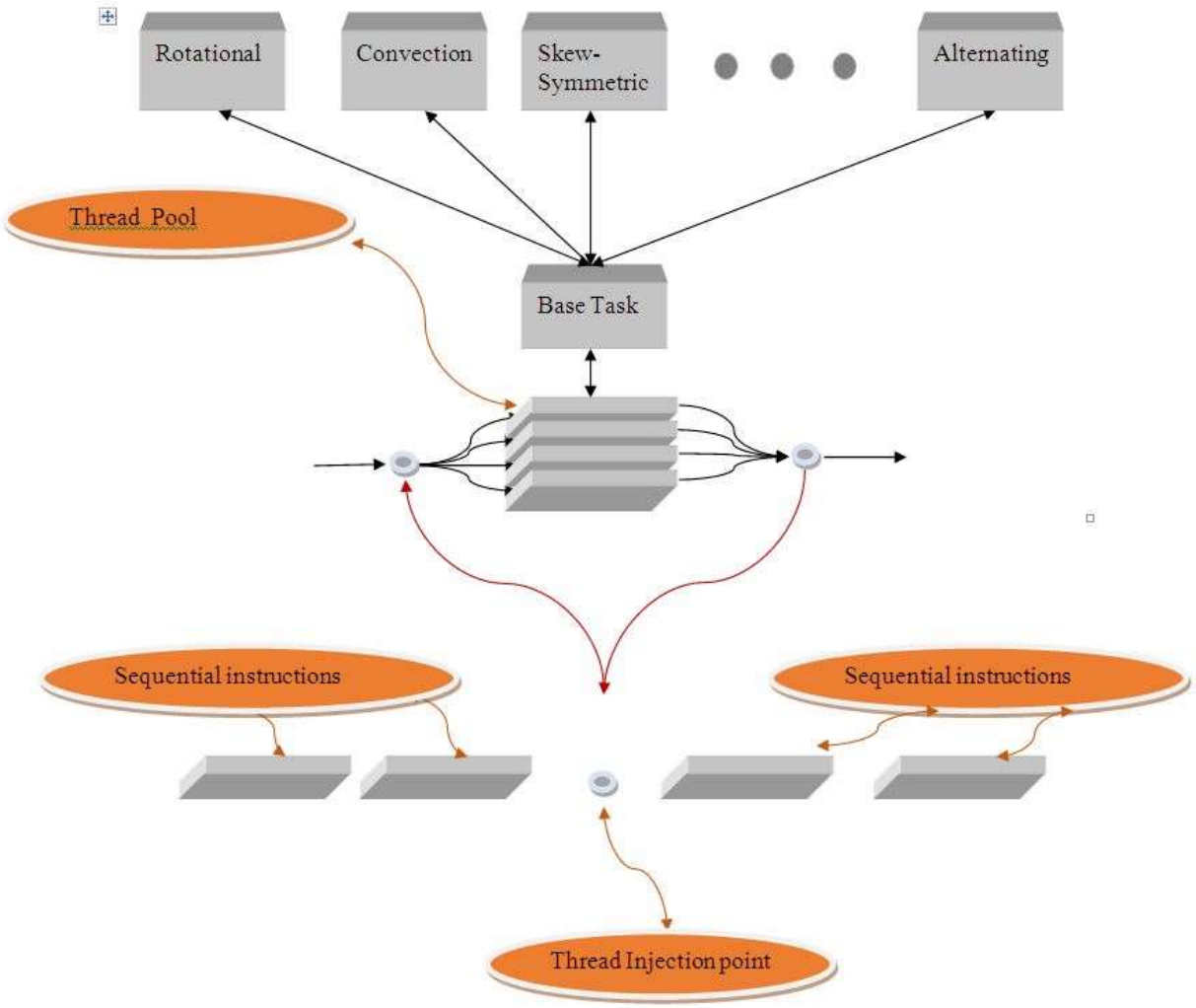


Figure 4.2 The parallelisation paradigm showing thread injection and the simulation process. At the foot of the diagram is a list of sequential instructions. The thread injection overwrites one of them. The thread pool takes control over one of the subsequent steps and executes it in parallel. Then control is returned to the next sequential instruction

The existing *Channelflow* program is executed serially, and an objective of our work is to identify those sections of the code that would benefit from parallelisation. Hence, we have

modified it so that it continues to run serially until it is signalled that those regions which would benefit from parallelisation have been reached. At this point, threads that control the computations are injected. The number of threads created by the thread pool depends on the size of the grid. In general, the more extensive the network, the more threads are created. Here we observe that when the computational fluid dynamics system consists of a small number of nodes, the number of threads is limited to reduce computational overheads. As a result, our approach automatically responds to the size of the problem and memory is dynamic; i.e. it is allocated on a needs basis.

To fulfil the described functionality, we create the following data functions:

- **CreateThreads** - will set up and start as many threads as specified by the parameter m_nmbThreads;
- **Run** – will unleash threads and let them run;
- **DestroyThreadPoo**l - will stop all threads running and remove the object of the Thread Pool from the system;
- **SetLimits** – specifying the location of the boundary and boundary conditions;
- **GetInstance** - return pointer for the object of the ThreadPool; only one object of this class is created;
- **WaitToComplete** - synchronize all threads completion. It makes sure there will not be a return from the ThreadPool until all threads have finished their computations. After all, tasks are finished all threads remain in the waiting mode; and **WakeUp** - let all threads know that a new execution task has arrived and they need to proceed with calculations. When a new task arrives, the thread pool wakes up its threads, and they begin to execute this task in parallel.

4.1.1.1 THREAD POOL ORGANIZATION

Below is a diagram that shows how the pool is organised. The highlighted box represents the ThreadPool class and on a chart is shown its collaboration with other objects.

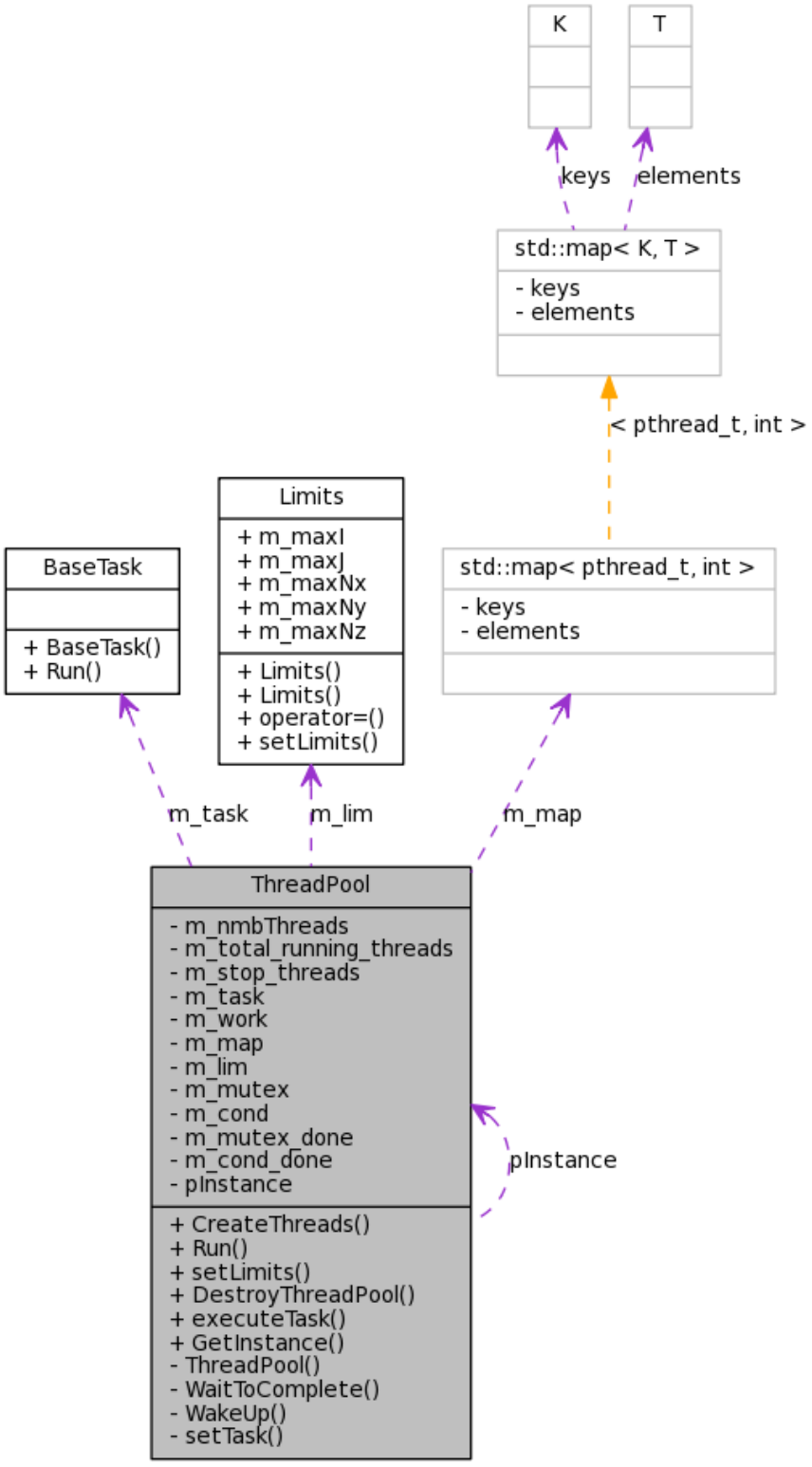


Figure 4.3 Thread Pool Organization diagram. Each box on the diagram represent a class and is divided into two parts; the top one lists the class variables and bottom one lists class public member functions.

The following class data members are implemented:

- **m_nmbThreads**. Variable to hold the number of threads that are going to be used by the Thread Pool;
- **m_total_running_threads**. A mutually exclusive variable to hold some running threads. When a particular thread has finished executing its task, this variable is reduced by one. Ultimately, m_total_running_thread is used to notify the thread pool that all threads have finished execution and the thread pool can return control to the sequentially running part of the application;
- **m_stop_threads**. Boolean variable used to control flow of running's threads;
- **m_task**. Base task pointer used as an interface to one of the CFD tasks;
- **m_work**. Dynamic array of Boolean flags. The size of the array is m_nmbThreads; Each flag represents a thread available for new work or still doing its current work;
- **m_map**. Map of threads handles;
- **m_lim**. The object of limits class, to hold the limits of the geometry domain;
- **m_mutex**. The instance of the class mutex to synchronise the data on threads access and signalling data engine;
- **m_cond**. Thread pool condition variable responsible for synchronisation of threads execution;
- **m_mutex_done _t m_cond_done** . Mutex and conditional variable responsible for control over signalling that execution of the job by thread is done; and

- **pInstance**- Pointer to an instance of the ThreadPool object. ThreadPool is a singleton class and pInstance use to control that only one object of this class created.

Threads are always running, so there is no overhead to create and start them. A thread pool communicates with the sequential part of the program by the signals, making this approach very flexible and dynamic. When the thread pool receives a request for the new task to be carried out, each thread independently starts traversing its sub-domain and collects the corresponding flowfield variable f(nx, ny, nz, i). When a thread has finished its calculation, it signals the pool that it has completed its task. When all threads have finished their work, the pool collects data from the threads and put threads in the waiting mode until a new task arrives. This technique

appears to be very elegant because it does not require the original source code to be changed. The thread pool class is instantiated only once during the startup. Then we inject our threads to parallelise the most time-consuming part of the simulation.

4.2 THREADPOOL CLASS REFERENCE

4.2.1 PUBLIC MEMBER FUNCTIONS

One of the aims of this research is to devise an object-oriented approach that facilitates users to easily and rapidly solve the Navier-Stokes equations. In meeting this goal, we specify the number of public member functions. These allow the user to specify the characteristics of the system that is under investigation, but there is no necessity to have a deep knowledge of the architecture and details that lie behind the program. CreateThreads enables the user to nominate the number of threads that may act as working agents and that are ready to execute the task. Furthermore, by making use of public member functions, users can specify the physical characteristics of the system they are investigating; these include the physical size of the system and the properties of the fluid. Through the object setLimits (Limits &) users are also able to specify numerical parameters that govern the calculation, such as the number of nodes in each of the three spatial directions. In *Channelflow* the time st*ep is autom*atically re-calculated at each time to maintain the desired accuracy.

- void CreateThreads (int numOfThreads)
- void * Run ()
- void setLimits (Limits &)
- void DestroyThreadPool ()
- void executeTask (BaseTask *)

4.2.2 STATIC PUBLIC MEMBER FUNCTIONS

The integrity of the thread pool must be protected from incursions by the user. This is achieved by creating the ThreadPool as a singleton - only one object of this class can exist. This is accomplished by declaring the ThreadPool constructor as private. However, to let users access the ThreadPool object, we create the GetInstance public method, which will point the user to the

location of our ThreadPool class. This provides users with access to all public interfaces but prevents them from making unintentional errors: it offers one stand-alone thread pool that is dedicated to our common task of solving Navier-Stokes equations.

The ThreadPool function has only one address; however, each thread can make calls to GetInstance, and they may carry out their tasks on a distributed system.

static ThreadPool * GetInstance ()

4.2.3 PRIVATE MEMBER FUNCTIONS

A key motivation that underpins this work is a desire to make our approach very general and in one sense, not problem-specific. For this reason, we have adopted what might be termed a macro-management approach to handling threads. The thread pool dispatches threads to carry out their tasks utilising setTask (BaseTask) but it does not direct their actions in detail. The task of the ThreadPool is to 'wake up' the threads when they are required and synchronise their actions by waiting for them to complete the tasks. These are designated private member functions.

- ThreadPool ()
- void WaitToComplete ()
- void WakeUp ()
- void setTask (BaseTask *)

4.2.4 PRIVATE ATTRIBUTES

We have described the ThreadPool as macromanaging the threads. However, within the ThreadPool the threads themselves must be managed. For example, in Chapter 3, we briefly mentioned the idea of mutex, which is used to prevent two threads from corrupting common data. In this particular case, we use mutexes to protect data and conflict between conditional flags.

- int m_nmbThreads
- int m_total_running_threads
- bool m_stop_threads

- BaseTask * m_task
- bool * m_work
- map< pthread_t, int > m_map
- Limits m_lim
- pthread_mutex_t m_mutex
- pthread_cond_t m_cond
- pthread_mutex_t m_mutex_done
- pthread_cond_t m_cond_done

4.2.5 STATIC PRIVATE ATTRIBUTES

We have noted that users must obtain access to the ThreadPool object using the ThreadPool::GetInstance method. However, it is important that users from outside the class cannot accidentally corrupt this memory location. We can ensure that this is the case by exploiting the fact that we declare pInstance is static private which guarantee that static ThreadPool * pInstance = NULL.

4.2.6 CONSTRUCTOR AND DESTRUCTOR

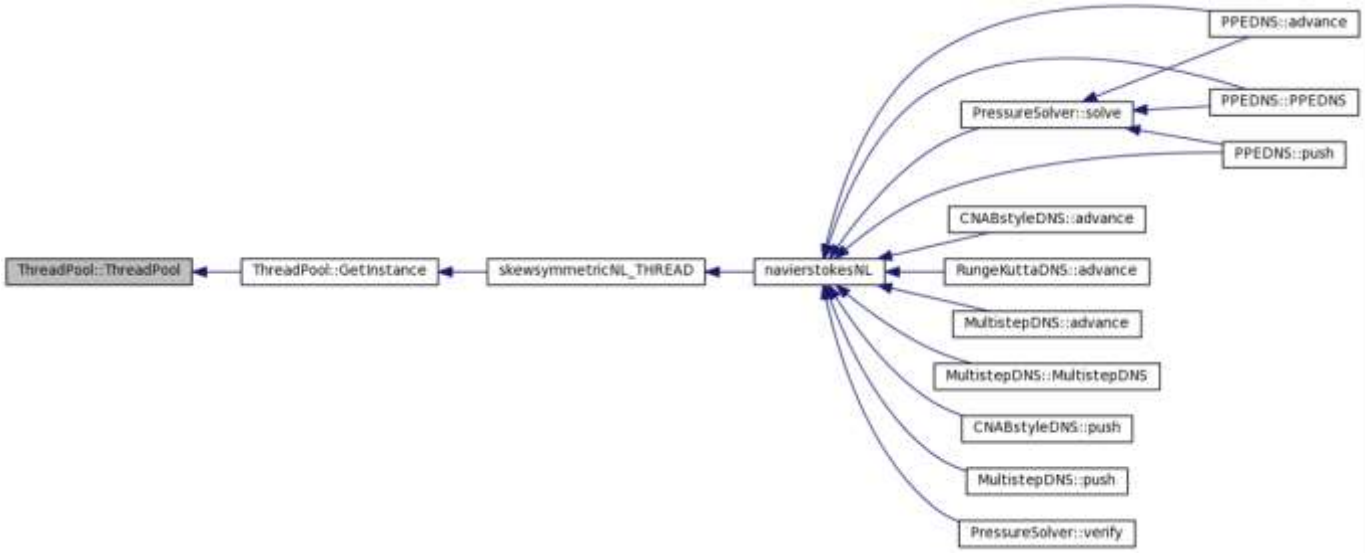
Parallelisation provides us with the capacity to solve computationally significant and substantial problems, possibly using several computers simultaneously. This entails making use of dynamic resources (see Chapter 3) but imposes responsibilities on programmers to free up these resources after the program has been terminated, which can involve invoking considerable programming logic. In our case, we need to consider how we de-allocate all of the threads because otherwise, they would continue to run, and this is achieved through a destructor. We also need to think about how the ThreadPool variables are initialised, which is achieved using a constructor ThreadPool::ThreadPool (). When the Threadpool object has finished, we ensure that ThreadPool::~ThreadPool ().

Here we initialize the variables used by the object ThreadPool when it is instantiated.

```
413 {
414       m_task = NULL;
415       m_mutex=PTHREAD_MUTEX_INITIALIZER;
```

```
416       m_cond=PTHREAD_COND_INITIALIZER;
417
418       m_mutex_done = PTHREAD_MUTEX_INITIALIZER;
419       m_cond_done = PTHREAD_COND_INITIALIZER;
420
421 }
```

The ideas discussed above require that the ThreadPool constructor is declared as private. We use the data hiding attribute of the Object-Oriented Design here to create a singleton version of the class. These requirements are captured by the call graph below. It highlights the fact that the original *Channelflow* remains extant, but we implement thread injection to speed up the execution of the non-linear term in the Navier-Stokes equations.



4.2.7 MEMBER FUNCTIONS

4.2.7.1 void ThreadPool::CreateThreads (int *numOfThreads*)

The speed of execution is a *raison d'être* of parallelisation. With this in mind, we have developed the CreateThread function to take advantage of the properties of Map, which is a container which is a very efficient search algorithm used to find and manipulate a particular thread. Threads are created by ThreadPool and m_map keeps track of pairs of thread handles and thread numbers which allow for quick communication with threads.

References m_map, m_nmbThreads, m_stop_threads, m_work, and start_thread().

```
433 {
434       pthread_t handel;
435       m_work = new bool[numOfThreads];
436       m_nmbThreads = numOfThreads;
437       m_stop_threads = false;
438
439       for(int i = 0; i < numOfThreads; ++i)  //loop over  anticipated number of threads
440       {
441         m_work[i] = false;
442         pthread_create(&handel, NULL, &start_thread ,(void *) this );  //here threads are
created
443         m_map[handel] = i;
444         //m_handel.push_back(handel);
445
446       }
447 }
```

The following caller graph demonstrates inheritance and encapsulation of Object-Oriented principles that we have briefly discussed in the previous chapter. ThreadPool:: Run and BaseTask:: Run have the same signature so that the real call will be evaluated at the run time. However, ThreadPool class can be compiled and linked regardless, allowing us to separate it into different modules.

4.2.7.2 void ThreadPool::DestroyThreadPool ()

When the simulation is finished, and the program needs to stop running, we need to clear all dynamically allocated resources. Here we implement a method for the threadPool to properly destroy itself. This involves notification to all threads to stop running.
We use a broadcast method to deliver such messages to all running threads. As we work in a multithreading environment, we must protect our condition variable by mutexes (see Chapter 3) to ensure the delivery of this message to all the running threads.

References m_cond, m_mutex, and m_stop_threads.

```
466 {
467   m_stop_threads = true;
468   cout << "DestroyThreadPool called" << endl;
469    // protecting conditional variable by mutex
470    pthread_mutex_lock(&m_mutex);
471       pthread_cond_broadcast(&m_cond);
472    pthread_mutex_unlock(&m_mutex);
473
474 /*
475    for(int k = 0; k < m_handel.size(); k++)
476    {
477       pthread_join(m_handel[k], NULL);
478    }
479 */
480    //sleep(5);
481
482 }
```

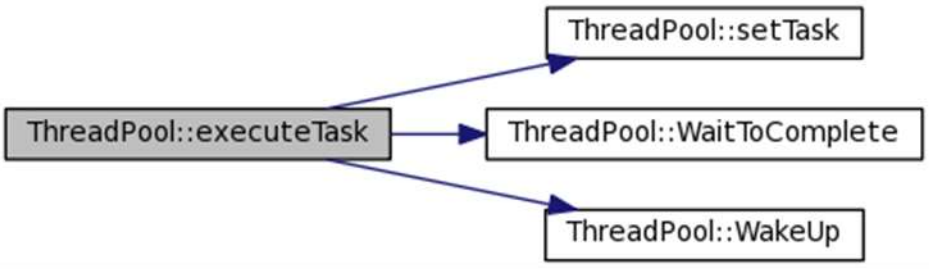
4.2.7.3 void ThreadPool::executeTask (BaseTask * *tsk*)

We have noted that we have designed the ThreadPool to be a macromanager. This is somewhat starkly exemplified by the following function. The job of the ThreadPool is to set the task, wake up the threads and wait for them to complete their tasks.

References setTask(), WaitToComplete(), and WakeUp().
Referenced by skewsymmetricNL_THREAD().

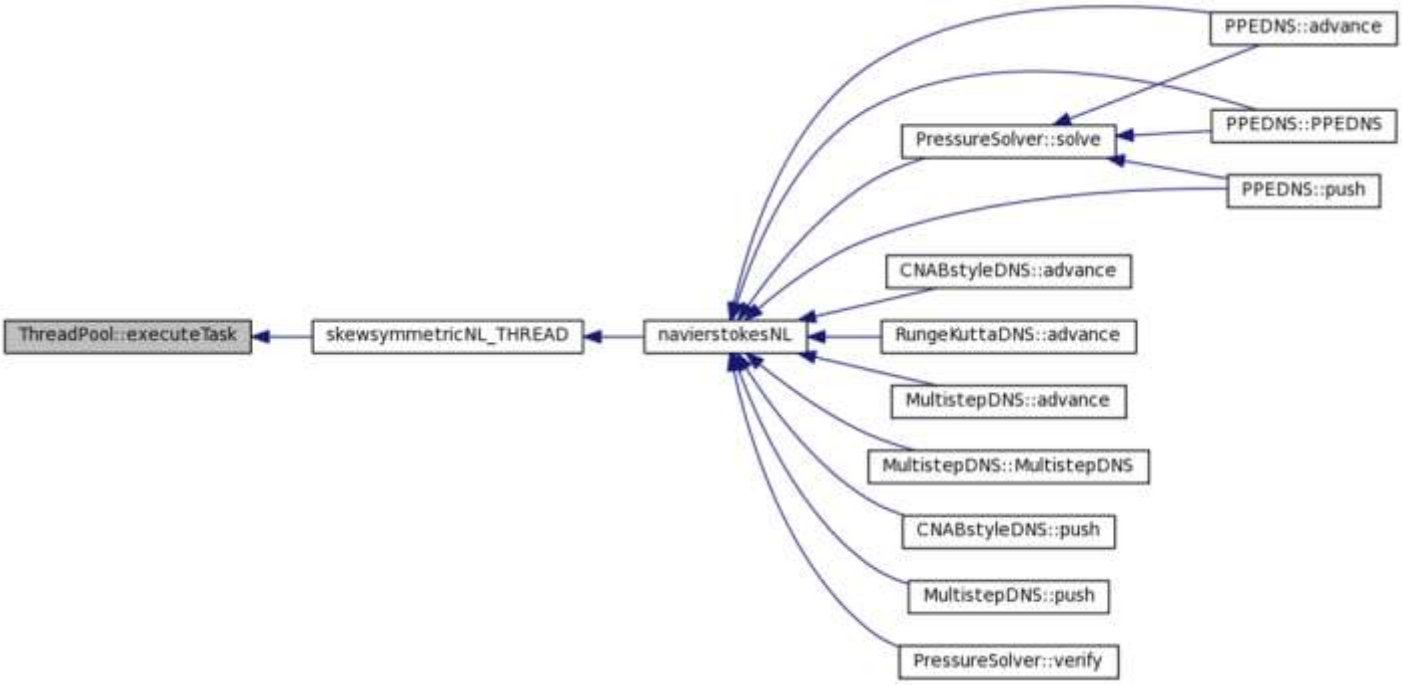
```
491 { //steps to make task executed
492   setTask(tsk);
493   WakeUp();
494   WaitToComplete();
495
496   delete tsk;
497 }
```

Below is the caller graph for this function:



As we can see in this diagram, the ThreadPool::executeTask has only managed the sequence of operations and does not have detailed knowledge of the duties of the task. This design is targeted to separate the core functionality from the plugin functionality. This makes the ThreadPool a stand-alone service. In the next graph, we see how detailed physics of fluid dynamics processes propagates in the core. This occurs during run time.

Below is the caller graph for this function:



4.2.7.4 ThreadPool * ThreadPool::GetInstance () [static]

ThreadPool:: GetInstance is a public interface to access the location of the ThreadPool object

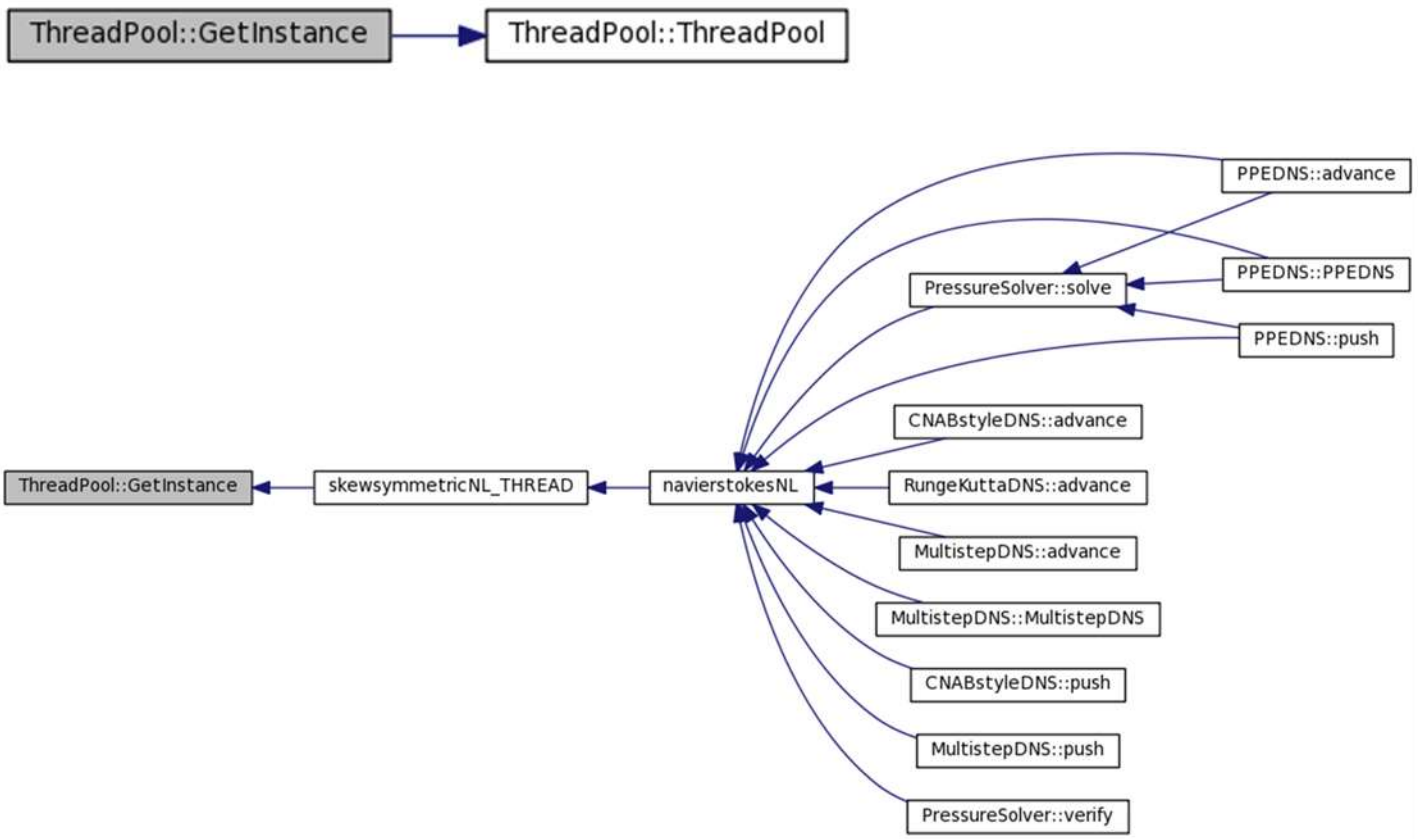
which was created by the private constructor. We briefly touched on encapsulation in the previous chapter, and here we present a detailed implementation. This element of our design guarantees that only one object of ThreadPool can exist.

The graph below shows how extensively GetInstance is called by all other classes:

References pInstance, and ThreadPool().
Referenced by skewsymmetricNL_THREAD().

```
423 {
424       if (pInstance== NULL)
425       {
426             pInstance = new ThreadPool();
427       }
428             return pInstance;
429 }
```

Here is the call graph for this function:



4.2.7.5 void * ThreadPool::Run ()

This method again demonstrates our philosophy of segregating the management of the program logic from the physics. It illustrates how ThreadPool delivers the Run message to the threads and how the threads report back when they have finished their tasks. Also, it maintains information on the number of currently running threads. Hence, we can observe that Run by the ThreadPool has the role of macro managing; however, all of the physics is encapsulated in the Run method of the particular thread.

References m_cond, m_cond_done, m_map, m_mutex, m_mutex_done, m_nmbThreads, m_stop_threads, m_task, m_total_running_threads, m_work, and BaseTask::Run().

Referenced by start_thread().

```
529 {
530    for (;;) //stays in infinitive loop till get signal
531    {
532       pthread_mutex_lock(&m_mutex);
533        int tn= m_map[pthread_self()];
534        while ( !m_work[tn] && m_stop_threads == false)
535        {
536          pthread_cond_wait(&m_cond, &m_mutex);
537
538        }
539        pthread_mutex_unlock(&m_mutex);
540
541        //cout <<"IG: worker thread id=" << tn << " running " <<endl;
542
543         if(m_stop_threads) break;
544
545         if(m_task) m_task->Run(tn,m_nmbThreads);
546         m_work[tn]= false;
547
548       pthread_mutex_lock(&m_mutex_done);
549          m_total_running_threads--;
550          pthread_cond_signal(&m_cond_done);
```

```
551             //cout << "IG: tn=" << tn << "finished m_total_running_threads="
<<m_total_running_threads <<endl;
552         pthread_mutex_unlock(&m_mutex_done);
553       }
554
555
556   return NULL;
557 }
```

Below is the caller graph for this function:

On this graph, we can see that ThreadPool:: Run and BaseTask:: Run have the same signature.

This architectural design demonstrates the inheritance principle, and we use it here to create dynamic calls which will lead us to execute a vast number of instructions.

Here is the caller graph for this function:

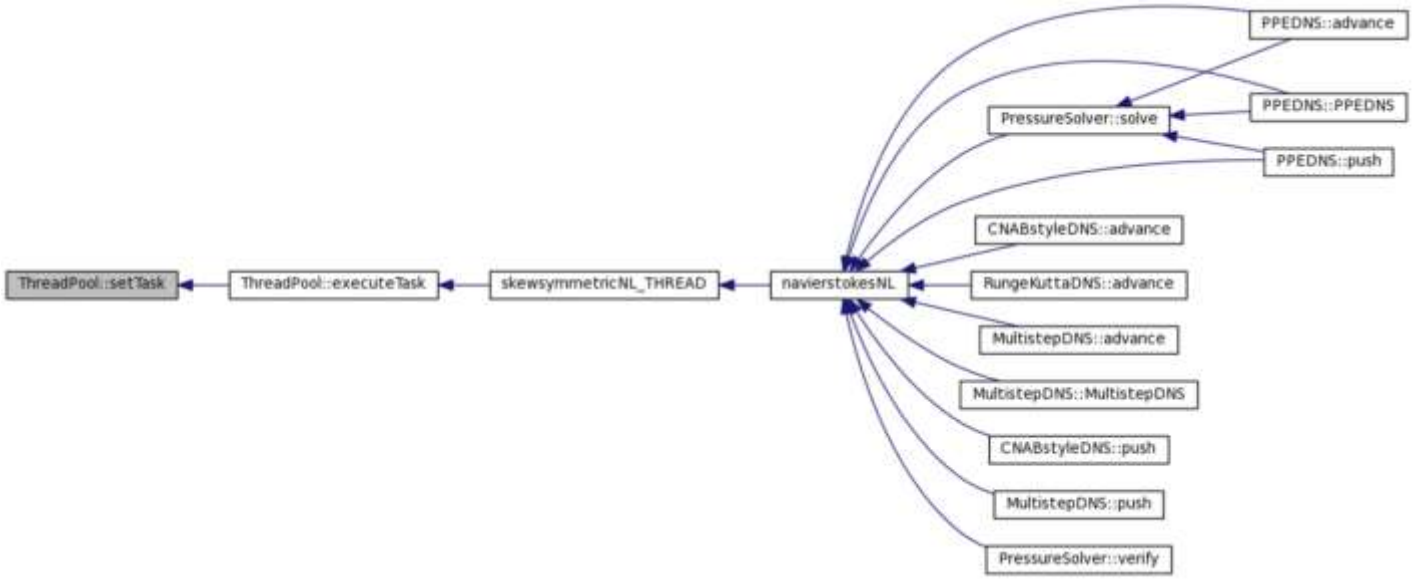
4.2.7.6 void ThreadPool::setTask (BaseTask * *tk*) [private]

References m_task.

Referenced by executeTask().

```
486 {
487   m_task = tk;
488 }
```

Below is the caller graph for this function:



4.2.7.7 void ThreadPool::WaitToComplete () [private]

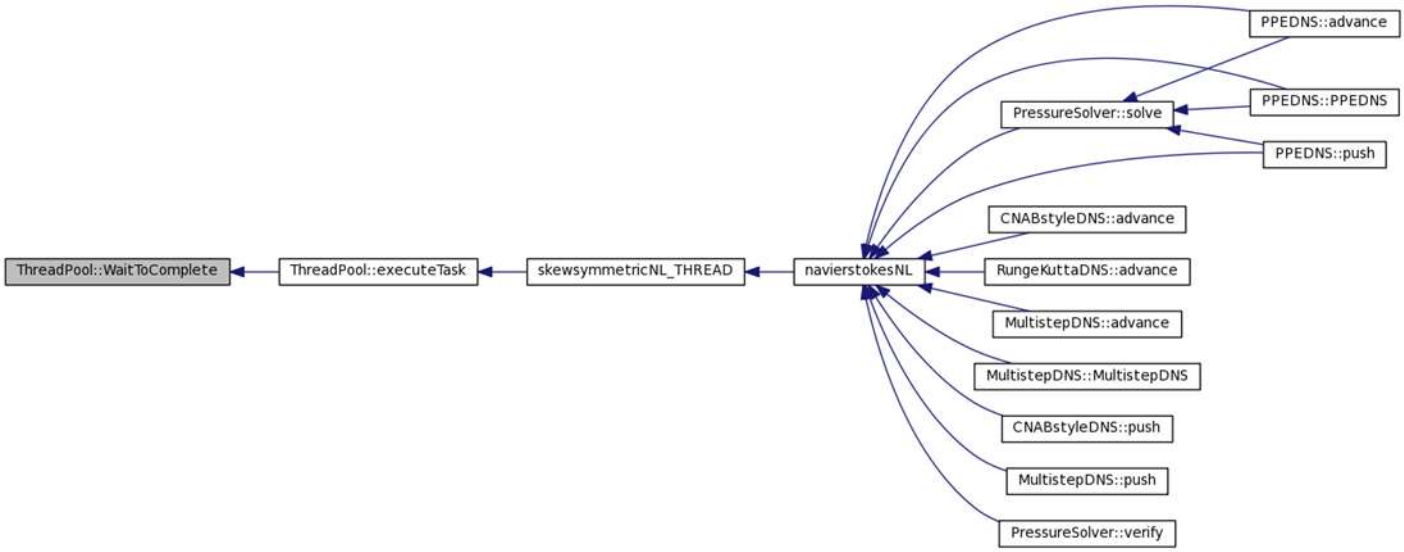
This is the private method of the ThreadPool class. According to the previous chapter, it should not be accessible from outside the object. The main idea of this approach is to use software interrupts to obtain signals from the thread and let it run. It is achieved by using condition wait, and this resource is protected by mutexes, as can be observed in the following snippet of code:

References m_cond_done, m_mutex_done, and m_total_running_threads.
Referenced by executeTask().

```
503 {
504     for(;;) //stay in infinitive loop till get the signal
505     {
506       pthread_mutex_lock(&m_mutex_done);
507          while(m_total_running_threads > 0)
508          {
509             pthread_cond_wait(&m_cond_done, &m_mutex_done);
510             //cout << "IG wake up as thread finised " <<endl;
511          }
512       pthread_mutex_unlock(&m_mutex_done);
513       if(m_total_running_threads <= 0) break;
514      }
```

```
515 }
```

Below is the caller graph for this function:



4.2.7.8 void ThreadPool::WakeUp () [private]

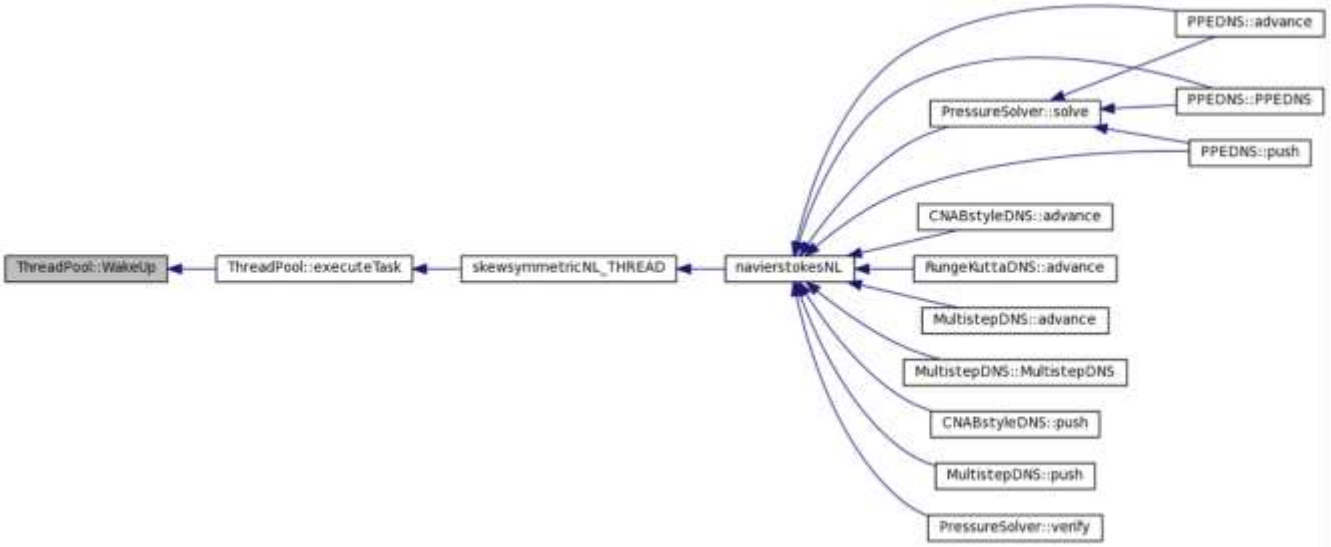
References m_cond, m_mutex, m_nmbThreads, m_stop_threads, m_total_running_threads, and m_work.

Referenced by executeTask().

```
450 {
451
452     m_total_running_threads = m_nmbThreads;
453     m_stop_threads = false;
454
455     for(int k = 0; k < m_nmbThreads; k++)
456     {
457       m_work[k] = true; //set the flag
458      }
459
460      pthread_mutex_lock(&m_mutex); //protect conditional variable
461      pthread_cond_broadcast(&m_cond); // broadcasting
462     pthread_mutex_unlock(&m_mutex); // relise mutex
463 }
```

Below is the caller graph for this function:

Here we can see again our ThreadPool object is stand alone but it retains a connection with all *ChanelFlow* objects.



As illustrated, the thread pool class does not have a detailed knowledge of the instructions to be executed; it is just a carrier for them and makes the object of this class a very flexible tool to create an injection and take over subsequent calculations by processing them in parallel.

Thread pool has only two interface methods – setTask and ExecuteTask – which will supply information to the pool about what exactly they need to execute.

Now we have to design an interface which we are going to use to perform the calculation of the convection part of Navier-Stokes equations using different algorithms. For this purpose, we create a Base Task class. This is an abstract class, and there are no objects that can be created. Instead, we are going to derive our skew-symmetrical form, divergence form, convection form, and rotational form algorithms. Instead, real job classes will be derived from this Base Task class, making it an interface to supply accurate information to the Thread Pool.

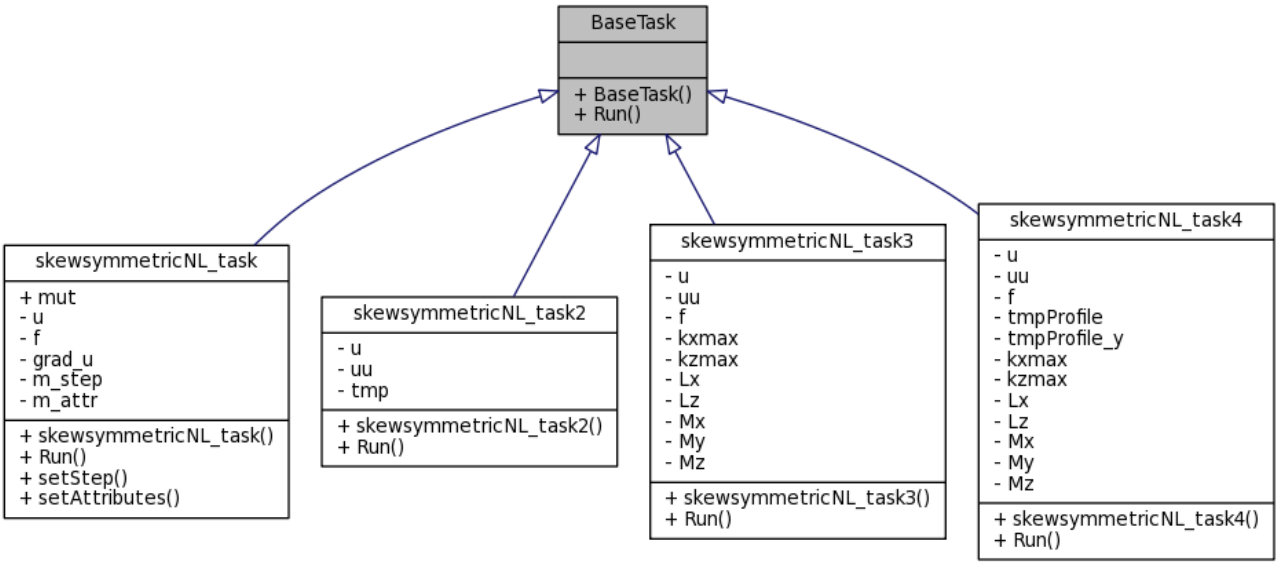


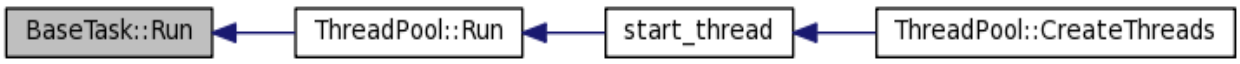
Figure 4.4 Diagram to show inheritance used by Base Task class to establish communication with concrete algorithms to calculate the non-linear part of Navier-Stokes equations.

In Figure 4.4, the rectangles below the Base Task box represent specific classes. Objects of these classes are instantiated during the run time. The upper part of each box represents the private part of the class where the specific logic for physical calculations is implemented. The bottom part of those boxes represents the public member function parts of the classes.

4.2.8 MEMBER FUNCTION DOCUMENTATION

4.2.8.1 VOID BASETASK:: RUN (INT TN, INT THREADS) [VIRTUAL]

Note that each of these classes has a public method called Run(), and it has to be the same signature as a Run method of the Base Task. Below is the caller graph for this function:



Here we can see how CreateThreads trigger start_thread which follows ThreadPool: Run method, and subsequentially Run method of the BaseTask class.

4.2.8.2 LIMITS

The location and dimensions of the boundaries, the number of computation nodes and their spatial distribution form key information that must be provided by the user. This information is held in the Limits class. Objects of this class are designed to hold information regarding domain dimension, grid properties, and dimension limits. Also, the object of this class is to hold information regarding internal nodes and nodes that belong to the boundary.

4.2.9 ACCESSING THREADS

When the thread is created, it returns its handle. The Thread Pool contains the information on each running thread by storing its handles in the map container. In this way, the Thread Pool can manage running threads and, depending on the load, add or stop some of the threads. However, access to the running threads is hidden from the user of the thread pool class.

In the table below, we compare existing *ChannelFlow* (**Before**) code and our Thread Injection approach (**After**). Our approach does not only make calculation much faster but also makes code more concise and more comfortable to follow.

Before	After

//Code snippet of Channelflow implementation
 	
// Accumulate 1/2 u_j du_i/dx_j in f_i
for (int i=0; i<3; ++i)
for (int j=0; j<3; ++j) {
int ij = i3j(i,j);
for (int ny=0; ny<Ny; ++ny)
for (int nx=0; nx<Nx; ++nx)
for (int nz=0; nz<Nz; ++nz)
f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
}
//
	//Code snippet after thread injection
	
	class skewsymmetricNL_task:public BaseTask
	pool->executeTask(new skewsymmetricNL_task(u, f, grad_u));

```
==========================================
  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
  // would overwrite results already in f (and changing order of
div
  // and convex calculations would require an extra transform)


  FlowField& uu = tmp;


  //outer(u,u,uu);
   t = clock();


  for (int ny=0; ny<Ny; ++ny) {
    for (int nx=0; nx<Nx; ++nx)
      for (int nz=0; nz<Nz; ++nz) {
        Real u0 = u(nx,ny,nz,0);
        Real u1 = u(nx,ny,nz,1);
        Real u2 = u(nx,ny,nz,2);
        uu(nx,ny,nz,0) = u0*u0;
        uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
        uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
        uu(nx,ny,nz,4) = u1*u1;
        uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
        uu(nx,ny,nz,8) = u2*u2;
      }
  }
for (int i=0; i<3; ++i) {
    int i0 = i3j(i,0);
    int i1 = i3j(i,1);
    int i2 = i3j(i,2);
```

```
   // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for
j=0,2
   for (int my=0; my<My; ++my)
     for (int mx=0; mx<Mx; ++mx) {
       int kx = u.kx(mx);
       Complex d_dx(0.0,
2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
       for (int mz=0; mz<Mz; ++mz) {
         int kz = u.kz(mz);
         Complex d_dz(0.0,
2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
         f.cmplx(mx,my,mz,i)
           +=
0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i
2));
       }
     }
   // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
   for (int mx=0; mx<Mx; ++mx)
     for (int mz=0; mz<Mz; ++mz) {
       for (int my=0; my<My; ++my)
         tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
       diff(tmpProfile, tmpProfile_y);
       for (int my=0; my<My; ++my)
         f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
     }
  }
```

Here we demonstrate how the paradigm discussed is applied to our CFD problem.

Line class skewsymmetricNL_task: public BaseTask is defined as a new type that is derived from the BaseTask type. Here we use inheritance to make the pool->executeTask accept this object as an object of the expected type.
Line new skewsymmetricNL_task(u, f, grad_u) creates an instance of the skewsymmetricNL_task object. All complexity shown on the left-hand side of the diagram is now hidden inside the Pool->executeTask method.
Also, we see them here the separation of duties. The skewsymmetricNL_task object responsible for physics, is the pool->execute_task responsible for the execution of this physics which is defined in the skewsymmetricNL_task object.
pool->execute_task does not have detailed knowledge of the object it passes for execution; instead, it merely responds by letting some threads execute it.
This paradigm ensures that all CFD calculations are separated by independent tasks and create a responsive and fast means of simulation.

4.3 skewsymmetricNL_task Class Reference

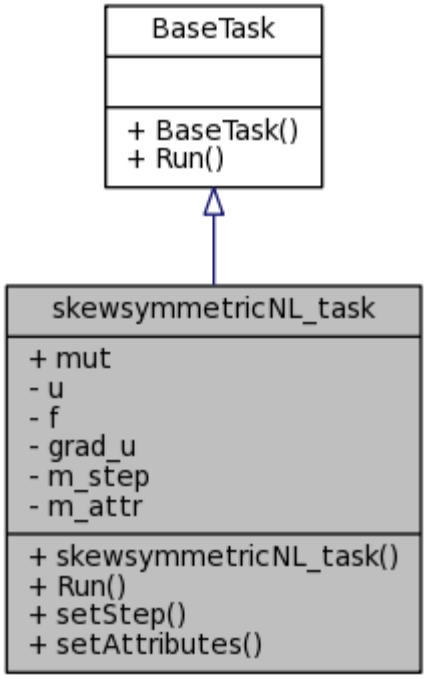
Here we arrive at the point of supplying an actual physics algorithm to our calculation of the non-linear part of Navier-Stokes equations. And this particular task is to implement the skew symmetrical scheme.

This algorithm is captured in the object and derives from the BaseTask class object through inheritance.

What we have briefly discussed in Chapter 3 is here demonstrated in detail.

This philosophy demonstrates how we separate Navier-Stokes calculations on independent parts from what is required for running things in parallel.

Below is the inheritance diagram for the skewsymmetricNL_task:
Here we can see how it inherits from the abstract BaseTask class, and we have specific Run() methods, which call through the Run() method of the BaseTask:



4.3.1 PUBLIC MEMBER FUNCTIONS

Here we design our interfaces where we can pass all necessary data and perform required physics calculations:

- skewsymmetricNL_task (const FlowField &_u, FlowField &_f, FlowField &_grad_u)
- void Run (int tn, int threads)
- void setStep (int step)
- void setAttributes (Attributes &attr)

4.3.2 PUBLIC ATTRIBUTES

In our design of this class we allocate the mutex variable to fulfill safe calculations which in case of multiple threads will need to access the same shared variable:

- pthread_mutex_t mut

4.3.3 PRIVATE ATTRIBUTES

In private attributes of this class we keep variables to fulfill our housekeeping calculations needed to maintain integrity with *ChannelFlow*:

- const FlowField & u
- FlowField & f
- FlowField & grad_u
- int m_step
- Attributes m_attr

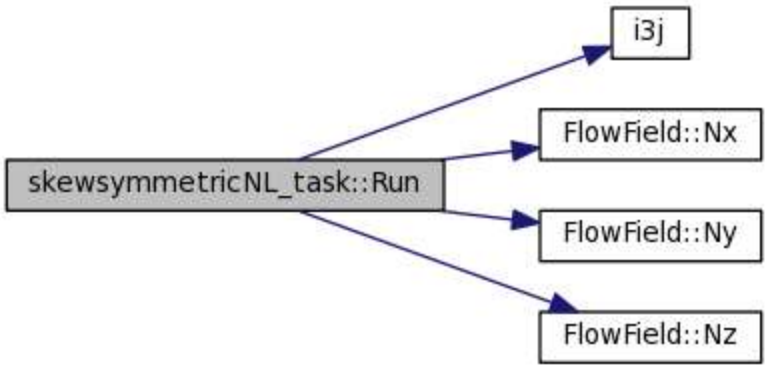
4.3.3.1 void skewsymmetricNL_task::Run (int *tn*, int *nThreads*) [virtual]

In this method we calculate flow field variable based on traversing the slice domain:

References f, grad_u, i3j(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), and u.

```
207 {
208  //cout << "Igor test skewsymmetricNL_step1 Run running " << endl;
209
210   int Ny = u.Ny()/nThreads;
211   int Nx = u.Nx();
212   int Nz = u.Nz();
213
214   int sty = Ny*tn;
215   int edy = Ny*(tn+1);
216
217   for (int i=0; i<3; ++i) //loop over slice domain
218     for (int ny=sty; ny < edy; ++ny)
219       for (int nx=0; nx < Nx; ++nx)
220         for (int nz=0; nz < Nz; ++nz)
221           for(int j=0; j <3; j++)
222           {
223              int ij = i3j(i,j);
224              f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
225           }
226
227
228 }
```

Here is the call graph for this function:



4.3.3.2 void skewsymmetricNL_task::setAttributes (Attributes & *attr*)

In our design, all physical attributes pass in as a reference to an instance of the Attributes class object.

This allows us to achieve several targets. Firstly, we can pass to the skewsymmetricNL_task all physical data in one go. Secondly, we again encapsulate this process and can easily add new data by altering the Attribute class. Note that this change will not require any alteration in theskewsymmetricNL_task::setAttributes method.

References m_attr.

```
202 {
203   m_attr = attr;
204 }
```

4.3.3.3 void skewsymmetricNL_task::setStep (int *step*)

References m_step.

```
197 {
198   m_step = step;
199 }
```

4.4 PARALLEL FFTW

Running FFT in parallel is another trigger that we use to speed up simulation processes.

Using spectral methods in CFD require that we apply Fourier transformation to do calculations in spectral space. Later on, it requires doing a back Fourier Transform to a physical space.

Equations 4.5 and 4.6 shows what exactly needs to be calculated.

The forward FFTW transform a complex array X of size n to an array Y.

$$Y_i = \sum_{j=0}^{n-1} X_j \, e^{-2\pi i j \sqrt{-1}/n} \tag{4.5}$$

The backward transform compute

$$Y_i = \sum_{j=0}^{n-1} X_j \, e^{2\pi i j \sqrt{-1}/n} \tag{4.6}$$

This process is very computationally expensive and requires about n^2 operations. The Fast Fourier Transform is an effective algorithm for computing the Discrete Fourier Transform and is significantly faster as it requires only n*log(n) operations.

FFTW (Frigo et al., 1998) is an open-source implementation of FFT. At the moment, it is still considered the fastest implemented FFT algorithm.

FFTW has inbuilt multithreaded capabilities which make encapsulating it in DNS code relatively easy.

4.5 TIME MEASURE IN PARALLEL ENVIRONMENT

The raison d'être of this element of our work is to speed up the execution of *Channelflow*, although it should be realised that this is merely an exemplar of our approach. However, this gives rise to an important question: how do we measure time in a parallel environment. Generally, one of the main reasons to use parallel processing is to make a program run faster. To parallelise a program or algorithm, we need to know which of its parts takes the most computational time. The CPU time is used as a parameter to measure performance, but it can be only used in sequential processes. In a parallel world, it does not work. We need to measure wall clock time, including communications and synchronisation overheads.

There are some different performance testing tools available. Profiling allows us to gather statistics about the time spent by applications in various program modules. Typically, it switches on by rebuilding applications using parameters for profiler options. Then, when the program is run it generates a table where time spent in different functions is listed as well as times these features were called.

There is different time measurement which reproduces different clocks used in reflecting the performance of calculations:

- **Wall clock time** is the amount of time taken to execute code in user space. It is calculated as the sum of CPU time, I/O time and communication channel delay;
- **User time** is the time the CPU is busy executing code in user space;
- **System time** is the time the CPU is busy executing code in kernel space;
- **Idle time** is the time the CPU is not busy. Idle time measures unused CPU capacity; and
- **Steal time** is the time consumed by the operating system to execute, but was not allowed to by the hypervisor. Running the top command can produce two metrics to indicate steal time. Percent idle (%id) and %wa percent I/O wait (%wa). When (%id) is low, the CPU is working hard; however, when (%wa) is high the CPU is ready to run, but is waiting for I/O to complete it operation.

When a program runs in parallel, total CPU time for that program would be more than its elapsed real-time. However, the wall clock time may be significantly less.

4.5.1 RESULTS

We ran channel flow for default domain size Nx=64, Ny=65, Nz=32 and Re=4000 and did experiments with thread pool running 1, 2, 3 and 4 threads.

Our results presented in Figure 4.5 show significant improvement in the time of simulation using the proposed thread injection technique. We can observe that increasing the number of working threads allows the same simulation to perform in significantly less time.

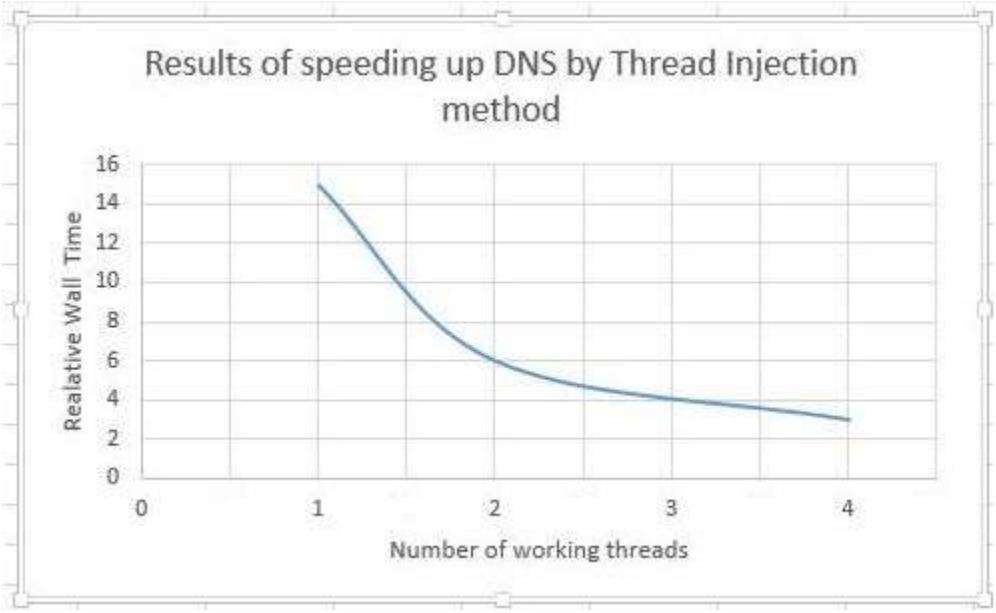


Figure 4.5 The speeding up of CFD Channelflow by adopting thread injection method

4.6 SUMMARY

One of the prime motivations of this research is to help scientists and engineers who are well versed in the serial world of computing to realise the advantages to be gained by parallelising their code. However, they will require a different mindset.

The method we have developed for DNS calculations has proved to be quite efficient and easy to implement. Thread injection replaces sequential code execution with some threads running in parallel. The thread pool class is a stand-alone object that can serve many CFD tasks.

The graph presented in Figure 4.5 shows the results of speeding up of simulation channel flow DNS using developed in this work thread pool and thread injection technique. We can observe here the significant speeding up of this simulation compared with sequential code implemented in *ChannelFlow*.

5 A PLATFORM THAT ACCEPTS SUB-GRID MODELS AS PLUG-INS TO ENABLE THE TESTING OF LES MODELS AGAINST DNS DATA

5.1 INTRODUCTION

The Johns Hopkins Turbulent Databases (JHTDB) is a catalogue of solutions of the Navier-Stokes equations. These solutions produced by direct numerical simulation (DNS) are accurate up to six decimal places. However, the solution is generated at 1024×1024×1024 grid points in space and 1024 time-samples containing 160 petabytes of information. The size of this database represents a severe obstacle to using it on a routine basis for practical analysis. An answer to this problem is to seek the application of 'database technology' in turbulence research and computational fluid dynamics (CFD). Direct numerical simulation **(DNS) of the Navier-Stokes equations resolves all of the flow structures that affect turbulent flows. However, in the case of LES, the Navier-Stokes equations are spatially filtered so that they are expressed in terms of the velocities of larger-scale structures. The rate of viscous dissipation is quantified by modelling the shear stress, and this process can lead to error.

Therefore, rapid testing and evaluation of models are necessary, and this is wholly associated with working with large sets of data. In this work, we present a computing platform that allows one to dynamically load LES models and quickly compare them to DNS results. The main idea permeating our methodology is that the core is defined as that which contains the 'know-how' associated with accessing and manipulating data, and which operates independently of a plugin. In our work, we presented an example demonstrating how users can examine the accuracy of LES models and get results almost instantly.

5.2 PROBLEM DESCRIPTION

Flow structures in turbulent flows span many orders of magnitude of length and time scales. They range from the length scale at which tiny eddies lose their coherence as their translational kinetic energy is dissipated into heat, up to eddies the size of which is related to that of the macroscopic system. The behaviour of the range of flow structures is captured by assuming that the fluid is a continuum, and they described by solving the Navier-Stokes equations. However, the limitations of computers restrict solutions of the Navier-Stokes equations to low Reynolds number flows in simple geometries. As alluded to above, in most practical situations these restrictions make it unfeasible to resolve features of turbulent flows on the smallest length and time scales. Engineers and scientists must, therefore, resort to empirical models of these small-scale phenomena that are expressed in simple mathematical terms. The models typically involve some form of averaging and approximations. Hence we must have some simple way of comparing their accuracy with the exact solutions of the Navier-Stokes equations.

These solutions are accurate to about six decimal places for mesh size is about 1024×1024×1024 grid points in space and 1024 time samples — that span the most massive flow structure. The entire space-time history of turbulence contains more than 10E12 data points, and users can access this data remotely using Web-servii interfaces. The JHTDB is a valuable source of information for comparison and evaluation models of turbulence.

However, the JHTDB database contains 160 petabytes (1.6×1017 B) of information, and this is a severe obstacle to using it routinely for practical analyses. A natural answer to this challenge is to seek the application of 'database technology' in computational fluid dynamics (CFD) and turbulence research. Turbulent flows are inherently unsteady and can have significant implications in many practical situations. For example, waves may give rise to substantial fluctuating forces on bluff bodies immersed in turbulent flows, and methods must be found to attenuate these effects. Flows through tree canopies, for example, are significant in determining the rate of exchange of gases such as water vapour and carbon dioxide with the atmosphere, and a good understanding of these phenomena is essential when studying climate change.

This research was inspired by a paper by Lee et al. (2015) in which challenges of working with massive data sets are described, and a call is made for the development of "database technology" in the area of computational flow dynamics. The aim is to seek automated ways of identifying

patterns and reduced-order descriptions, developing machine learning, and performing data mining and so on, to reduce a significant amount of data to be transmitted and processed.

In our work, we anticipate creating a platform using modular programming which allows LES models to be rapidly evaluated and dynamically loaded to compare against DNS results such as those available in the JHTDB. LES and DNS solutions are compared for turbulent flow with a Reynolds number based on the Taylor microscale, λ, of 433, and they demonstrate that refining the filter width results in more accurate solutions of the Navier-Stokes equations.

5.3 GOVERNING EQUATIONS

The flow behaviour of incompressible viscous fluids is governed by the Navier-Stokes equations expressed as

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = f_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x}\left(\nu \frac{\partial u_i}{\partial x_j}\right), \tag{5.1}$$

and the fluid must obey the conservation of mass that is represented by

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{5.2}$$

The idea that underpins large eddy simulation (LES) is that the behaviour of large-scale structures which occur in turbulent flows can be captured by spatially filtering the Navier-Stokes equations (Smagorinsky, 1963) and the dissipative structures modelled. .Mathematically, a spatial filtering operation using a kernel G is defined as:

$$\bar{\varphi} = \int G(\underline{x} - \underline{y})\varphi(\underline{y})d\underline{y} \tag{5.3}$$

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = \bar{f}_i - \frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x}\left(\nu \frac{\partial \bar{u}_i}{\partial x_j}\right) + \frac{1}{\rho}\frac{\partial \bar{\tau}_{i,j}}{\partial x_j} \tag{5.4}$$

The result of this operation is a similar system of equations, but now unknown variables are filtered which have characteristics of averaging over the filter width size.

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0 \tag{5.5}$$

The advantage of this approach is that the system of equations is considerably reduced compared with the system required for DNS. However, the LES system of equations is not closed as it contains an extra term. A second issue arises with averaging techniques in general; namely, there is a loss of information. As a result, large eddies do not contain all the information that is required to compute the future of a given flow. To resolve these issues several subgrid-scale models were introduced. One of the earliest models was proposed by Smagorinsky (1963) that assumes the dissipation of energy is described by using a grid cell as a filter. Alternative models have been proposed, each one trying to address some issues of existing models (You, 2007). This gives rise to the question of how they might be compared. If the 'true' flow field is known, then it can be used to develop an *a priori* test. In our work, we are going to use the results of Direct Numerical Simulation (DNS) collected in the Johns Hopkins Turbulence Database (JHTDB).

In this work, we present a simple-to-use platform that can be employed to compare the effectiveness of a wide range of LES models and compare them with DNS data. To date, researchers have tended to deal with relatively simple data. They compare results of calculations with experimental results or compare them with obvious analytical solutions or some simplified

2-D graphs. The situation dramatically changed when we needed to deal with the 4-D space of petabytes of data.

This work is motivated by the view that LES is formulated as a system of n equations with n + 1 unknowns. There are endless opportunities to supply the final equation. Considering the space that this equation describes is quite significant, and LES is a promising tool for the future, our approach creates a considerable number of opportunities for researchers as well as for developers. The software presented here will be in the public domain.

5.4 DESIGN AND IMPLEMENTATION OF IDEAS

The purpose of this research is to produce a highly efficient platform that is an easy and convenient tool for the scientific community to devise and test their sub-grid models against the results of DNS. Schematically it is presented in Figure 5.1 below.

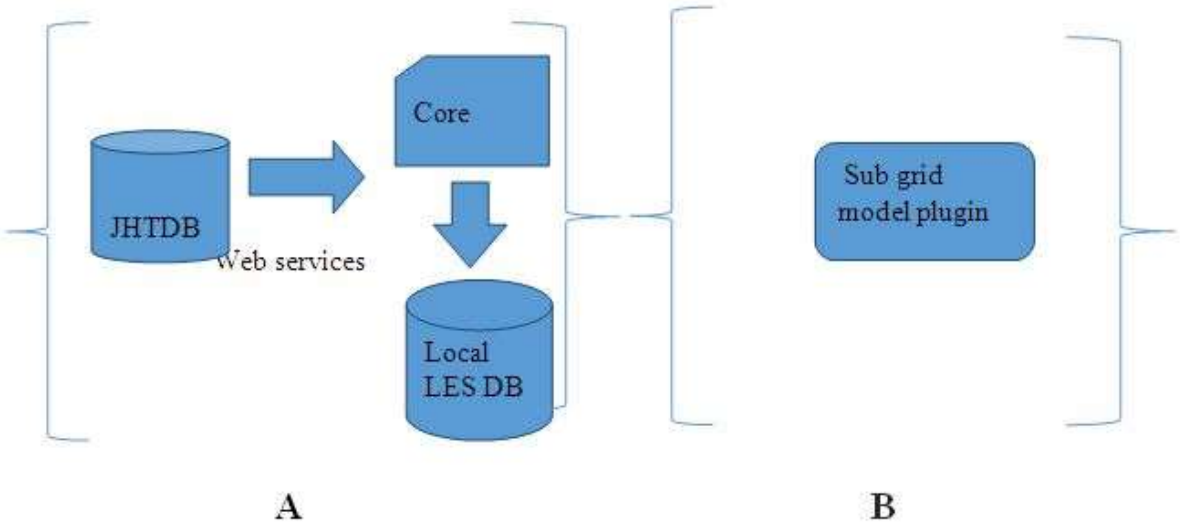


Figure 5.1 The platform comprises two components. A is known as the core, and B represents plug-ins that enable researchers to test the accuracy of their proposed LES models almost instantaneously

The system we have developed comprises two components, namely:

- The core A that can communicate with the Johns Hopkins Turbulent Database by web services and collect and store all relevant information on a local machine. It also includes

a range of numerical simulations, filtering operations and comparators that are not bound to any particular sub-grid model.

- A plugin component B where all specific model relations are set by the user.

A and B are two completely independent stand-alone modules.

The idea is that A can be extended by object B without having to rebuild A. One can think of A as a repository of knowledge that can be accessed by the user. The implementation of this idea provides a flexible platform that allows the scientific community to test and compare different sub-grid models without the necessity of writing extensive computer code.

C++ has proven to be a u useful language language for scientific calculations, and even if C++ does not provide the language support for plugin implementation, it nonetheless provides a sound basis for it. Firstly, we discuss the language capability that can be used to implement the platform we have designed. C++ pure virtual functions, abstract base classes, and interface classes form the foundation of our design implementation. The interface class is abstract; therefore, the compiler does not require a particular implementation of the class. Hence our module A does not have any linkage issues. The instantiation of the object class occurs during run time through the derived plugin classes.

5.5 KEY COMPONENTS

5.5.1 MEMORY MANAGEMENT.

The platform we have developed must handle large amounts of data. For this reason, memory management is crucial and requires special attention. Let us consider what and how the memory is used, and what options are available to the developer. The memory that an application uses is divided into four different areas:

- The code area, where the compiled program resides;
- The global variables area;

- The stack from which parameters and local variables are allocated; and
- The heap in which dynamically allocated variables reside.

The first two options are not very relevant because the code area is usually quite small and global variables are rarely used in the development of modern software. The stack and the heap are where most memory is located.

5.5.1.1 THE STACK

The temporary variables created by each function stored in the memory segment are called the stack. The stack memory is managed and optimised by the CPU. When we declare a new variable in the function or method, it is "pushed" onto the stack. When the function exits, all of the stack variables are freed, and that region of the memory becomes available for other stack variables.

The stack memory is managed by the operating system, which makes it more attractive to use. This memory does not need to be allocated by hand or released once it is no longer required. The operating system deletes this memory when variables have gone out of scope. Stack memory is quite fast and usually organised very efficiently. However, the size of the stack memory is restricted by the operating system (Elke, 2004).

5.5.1.2 THE HEAP

The heap – also known as the “free store” – is a large pool of memory used for dynamic allocation. This memory is not managed automatically by the operating system. It does not have a scope and requires manual creation. This memory needs to be free when it is no longer needed. If this process is not synchronised, the system may exhibit memory leak. Because the precise location of the memory allocated is not known in advance, it has to be accessed indirectly using pointers. Compared with the stack, the heap does not have size restrictions on variable size, from those arising from the obvious physical limitations imposed by the hardware (Wilson et al., 1995).

5.5.1.3 STACK VS HEAP

DNS and LES require manipulation of large amounts of data; hence although we have to manage the heap memory, it is a promising option for our project. We will describe later how exactly the heap is used in our platform.

5.6 DATABASE ENGINE

Our design aims to separate the core (know-how system) and the ability to use plugins with different LES models to test.

Our philosophy is based on the fact that the John Hopkins database is difficult to use for practical applications. Instead, we are going to implement our database, see Figure 5.1.

This database will have a built-in the two-way communication process. The first one is communication with the John Hopkins database and collecting DNS solutions, Then our computational engine will build filtering solutions and them in local DB. The second one is a feather communication with plugins to test different LES models. This is how knowledge will be built inside our database.

This dictates the unusual requirement for the database engine. Rather than be a stay alone server like MySql or SqlServer, which facilitate a lot of requests from clients, our database should be inbuilt into the process, and it has to be very light and fast. All of the above considerations led us to choose the SQLite engine (Haldar, 2015).

Compared to other database engines, SQLite is a server, not a client. It is a fast, light and reliable open-source library. SQLite has an interface with C and C++ and is thread-safe, which opens the door for massively parallel transactions. SQLite did not need a stand-alone server process and was embedded in a current working application. SQLite reads and writes directly to ordinary disk files. It is extremely fast, and this is why it employed in applications of many well-known users including Apple, Airbus, Sun Microsystems and Skype. There are many other users, but they have not all been identified because of the open-source nature of the SQLite. However, its popularity, and therefore its implied reliability, have prompted us to use this database engine (Haldar, 2015).

5.7 DATABASE CLASS REFERENCE

According to the philosophy of Object Oriented Principles, we designed our class as a separate entity with member functions which should make it atomic.

Here we present our design for our LES database class.

5.7.1 PUBLIC MEMBER FUNCTIONS

- Database (const char *filename)
- ~Database ()
- bool open (const char *filename)
- table query (char *query)
- void close ()
- void begin ()
- void commit ()
- void end ()

5.7.2 PRIVATE ATTRIBUTES

- sqlite3 * database

5.7.3 CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

Database::Database (const char * *filename*) // *this is the constructor code*

References database, and open().

```
5 {
6       database = NULL;
7       open(filename);
8 }
```

Below is the call graph for this function:

In this graph, we see that open database requires constructor of a Database call, an object of the Database class instantiate.

Database::~Database ()

// *This is the destructor code. We are not allowed to let the database object be automatically destroyed,*

// so we leave the lines below empty.

```
11 {
12 }
```

5.7.4 MEMBER FUNCTION

The database state is changed by the logical instructions of operation called transactions. Here in our design begin() and end() methods to specify start and end of the transaction.

void Database::begin ()

References database.

Referenced by core::createLES_DB().

```
23 {
24   sqlite3_exec(database, "BEGIN TRANSACTION;", NULL, NULL, NULL);
25 }
```

void Database::close ()

References database.

Referenced by core::createLES_DB(), and main().

```
76 {
77        sqlite3_close(database);
78 }
```

void Database::commit ()

References database.

Referenced by core::createLES_DB().

```
32 {
33   sqlite3_exec(database, "COMMIT", NULL, NULL, NULL);
34 }
```

void Database::end ()

References database.

Referenced by core::createLES_DB().

```
28 {
29    sqlite3_exec(database, "END TRANSACTION;", NULL, NULL, NULL);
30 }
```

bool Database::open (const char * *filename*)

References database.

Referenced by Database().

```
15 {
16        if(sqlite3_open(filename, &database) == SQLITE_OK)
17                return true;
18
19        return false;
20 }
```

table Database:: query (char * *query*)

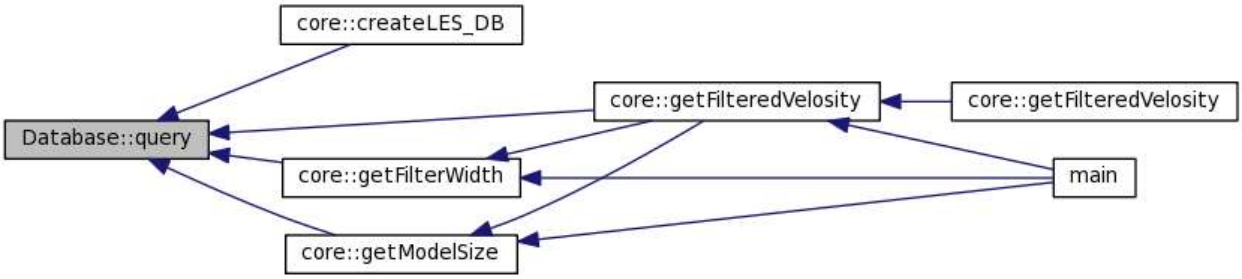
References database.

Referenced by core::createLES_DB(), core::getFilteredVelosity(), core::getFilterWidth(), and core::getModelSize().

```
39 { //the code below is a c++ wrapper to sql language for communicating with database
40        sqlite3_stmt *statement;              //prepare statement object
41        vector<vector<string> > results;  //declare matrix using stl containers
42
43        if(sqlite3_prepare_v2(database, query, -1, &statement, 0) == SQLITE_OK)
44        {
45                int cols = sqlite3_column_count(statement);
46                int result = 0;
47                while(true)
48                {
```

```
49                  result = sqlite3_step(statement); //start building results
50
51                  if(result == SQLITE_ROW)
52                  {
53                      vector<string> values;
54                      for(int col = 0; col < cols; col++)
55                      {
56                          values.push_back((char*)sqlite3_column_text(statement, col));
57                      }
58                      results.push_back(values); //populate stl container with results
59                  }
60                  else
61                  {
62                      break;
63                  }
64              }
65
66              sqlite3_finalize(statement); //release memory use by prepared statement
67          }
68
69          string error = sqlite3_errmsg(database); // prepare error message if found
70          if(error != "not an error") cout << query << " " << error << endl;
71
72          return results;
73 }
```

Below is the caller graph for this function:



In this graph, we see how the Database class correlated with the core class.

Any calls from the core like createLES_DB, getFilterWidth or getModel Size end up in a call to Database::query.

Member Data Documentation

sqlite3* Database::database [private]

Referenced by begin(), close(), commit(), Database(), end(), open(), and query().

5.8 FAST FOURIER TRANSFORMATION

Results of DNS and LES calculations form a 3-D field of N3 double-precision values in physical space where N is the domain size. To compare such broad groupings, we have to develop a comparator operator. This operator is based on comparing spectra populated in Fourier with frequency domain space. Spectra calculation is computationally intensive and requires efficient algorithms to perform Fast Fourier Transformation. To achieve this, we have chosen the FFTW open-source library developed at MIT by Frigo and Johnson (Frigo et al., 1998).

Our choice of FFTW was motivated by its performance which is superior to other available FFT software and is widely used in many scientific applications.

In the next chapter, we will have discussed in detail how this was done using the Edward supercomputer.

6 EDWARD HIGH-PERFORMANCE COMPUTER

6.1 INTRODUCTION

The project has been built, developed, tested and run on the Edward High-Performance Cluster based at the University of Melbourne. It executes commands that are about three orders of magnitude higher than personal computers. It has about six orders more RAM than a personal computer. But its main advantage is that it is based on the GNU/Linux operation system which renders it compatible with any other types of Linux operation system (Strazdins,2012; Galassi, 2009).

6.2 THE CORE CLASS AND ITS MEMBERS

CORE CLASS REFERENCE

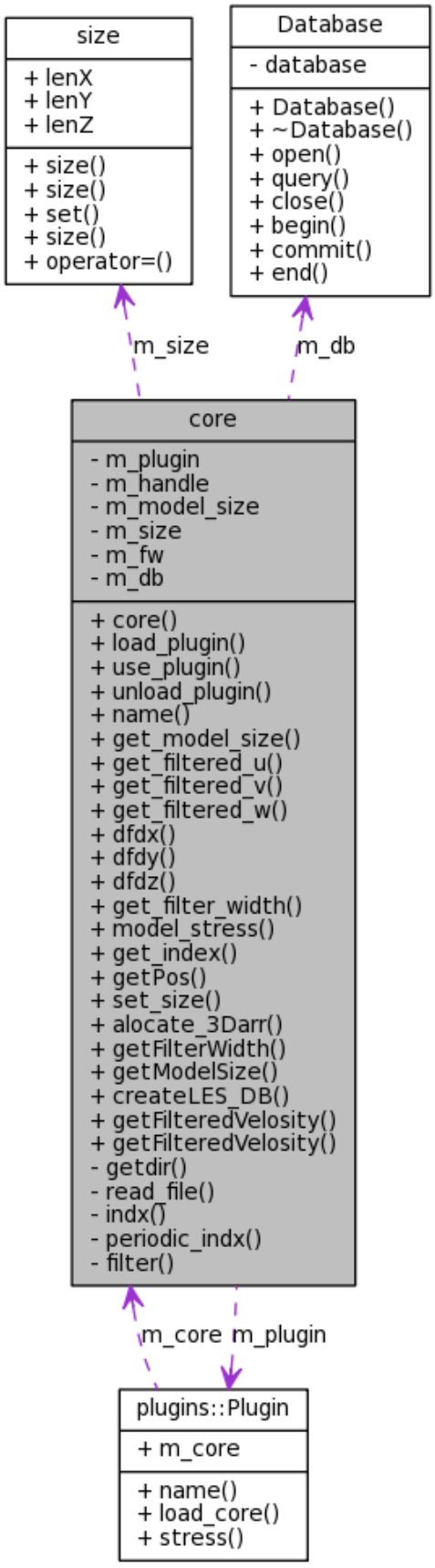

size
+ lenX
+ lenY
+ lenZ
+ size()
+ size()
+ set()
+ size()
+ operator=()
Database
- database
+ Database()
+ ~Database()
+ open()
+ query()
+ close()
+ begin()
+ commit()
+ end()
m_size
m_db
core
- m_plugin
- m_handle
- m_model_size
- m_size
- m_fw
- m_db
+ core()
+ load_plugin()
+ use_plugin()
+ unload_plugin()
+ name()
+ get_model_size()
+ get_filtered_u()
+ get_filtered_v()
+ get_filtered_w()
+ dfdx()
+ dfdy()
+ dfdz()
+ get_filter_width()
+ model_stress()
+ get_index()
+ getPos()
+ set_size()
+ alocate_3Darr()
+ getFilterWidth()
+ getModelSize()
+ createLES_DB()
+ getFilteredVelosity()
+ getFilteredVelosity()
- getdir()
- read_file()
- indx()
- periodic_indx()
- filter()
m_core
m_plugin
plugins::Plugin
+ m_core
+ name()
+ load_core()
+ stress()

Figure 6.1 Collaboration diagram for core class

6.2.1 PUBLIC MEMBER FUNCTIONS

Methods listed below will serve the role of interfaces to communicate with plugins and perform different LES simulations tasks:

- core ()
- void load_plugin ()
- void use_plugin ()
- void unload_plugin ()
- std::string name ()
- int get_model_size ()
- double * get_filtered_u ()
- double * get_filtered_v ()
- double * get_filtered_w ()
- double dfdx (int ind, order_t o, double *u)
- double dfdy (int ind, order_t o, double *u)
- double dfdz (int ind, order_t o, double *u)
- int get_filter_width ()
- void model_stress ()
- int get_index (size &pos)
- size getPos ()
- void set_size (size &s)
- double * allocate_3Darr (size &)
- int getFilterWidth (Database *db)
- int getModelSize (Database *db)
- void createLES_DB (std::string source_dir, filter_base &f, std::string out_dir)
- FilteredData getFilteredVelosity (point &p, Database *db)
- void getFilteredVelosity (LIST_POINTS &pl, LIST_DATA &ld, Database *db)

6.2.2 PRIVATE MEMBER FUNCTIONS

The private method of the class is not accessable from outside of the object and is intended to maintain internal object functionality:

- int getdir (std::string dir, std::string ext, std::vector< std::string > &vfiles, const std::string &optional="")
- int read_file (string &path, string &fname, size &sz, int iz, double *u, double *v, double *w)
- int indx (size &sz, int x, int y, int z)
- int periodic_indx (int size, int ind)
- double filter (point &, size sz, filter_base &f, double *v)

6.2.3 PRIVATE ATTRIBUTES

- plugins::Plugin * m_plugin
- void * m_handle
- int m_model_size
- size m_size
- int m_fw
- Database * m_db

6.2.4 CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

core::core () //default constructor , no data members set at this stage yet

```
41          : m_size(0,0,0),m_handle (NULL),m_fw(0),m_db(0)
42  {
43  }
```

6.2.5 MEMBER FUNCTION

6.2.5.1 double * core::alocate_3Darr (size & *sz*)

References size::lenX, size::lenY, and size::lenZ.
Referenced by createLES_DB().

```
395   {//dynamicly allocate space and build 3D array
396       double *arr=0;
397       arr = new double[sz.lenX * sz.lenY * sz.lenZ](); // dynamic memory allocation
398       if(arr == NULL) cout << "Can't allocate memory" << endl;
399       return arr;
400   }
```

Here is the caller graph for this function:

void core::createLES_DB (std::string *source_dir*, filter_base & *f*, std::string *out_dir*)

References alocate_3Darr(), Database::begin(), Database::close(), Database::commit(), Database::end(), filter(), filter_base::get_fw(), getdir(), indx(), Database::query(), read_file(), point::x, point::y, and point::z.

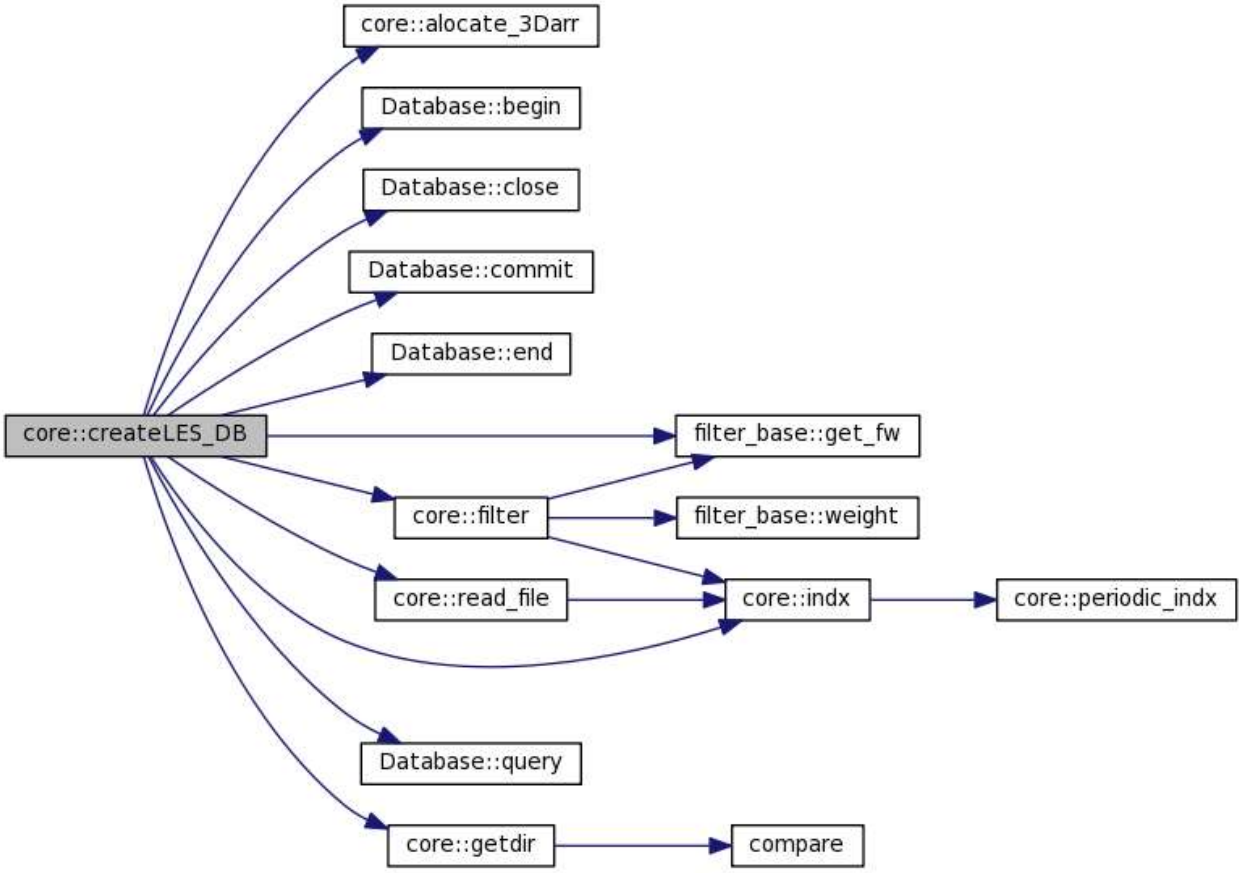
```
176   { // this is where is actual LES database been populated
177
178     struct stat buffer;
179     if( stat(out_dir.c_str(), &buffer) == 0)
180     {
181      cout << "out_dir " << out_dir << " exist. Override? (Y/N) " <<endl;
182      string choice;
183      getline(cin, choice);
184      while (choice != "n" && choice != "N" && choice != "y" && choice != "Y")
185      {
186         printf ("\nPlease enter Y (Yes) or N (No)\n");
187         getline(cin, choice);
188      }
189         if (choice == "Y" || choice == "y"){ remove(out_dir.c_str());}
190         else return ;
191     }
192
193
```

```
194
195     std::vector<std::string> files;
196     std::string ext="dat";
197     int nf = getdir(source_dir,ext,files);
198     size DNS_sz(nf,nf,nf);
199
200     double* u = alocate_3Darr(DNS_sz);
201     double* v = alocate_3Darr(DNS_sz);
202     double* w = alocate_3Darr(DNS_sz);
203
204
205     for(int z = 0; z < files.size();z++)
206     {
207       cout << "reading " << files[z] << endl;
208       read_file(source_dir,files[z],DNS_sz,z,u,v,w);
209     }
210
211
212     int fw=f.get_fw();
213     int LES_len = nf/fw;
214     int LES_cellNo = 0;
215
216     size LES_model_sz(LES_len,LES_len,LES_len);
217     size LES_cell_sz(fw,fw,fw);
218
219     char* path = (char*)source_dir.c_str() ;
220
221     Database *db =  new Database(out_dir.c_str());
222
223     db->query((char*)"CREATE TABLE source (filter_width INTEGER, LES_model_sz
INTEGER, path TEXT);");
224     db->query((char*)"CREATE TABLE data   (ind INTEGER PRIMARY KEY,
filtered_u REAL, filtered_v REAL , filtered_w REAL);");
225
226     stringstream sfw;
227     sfw<<fw;
228     stringstream slen;
229     slen<<LES_len;
230     string ch=",";
231     string en=" );";
232     string qvo="\"";
233
234
235     string insert_source="INSERT INTO source VALUES(";
236     insert_source += sfw.str();
237     insert_source += ch;
```

```
238     insert_source += slen.str();
239     insert_source += ch;
240     insert_source += qvo;
241     insert_source += source_dir;
242     insert_source += qvo;
243     insert_source += en;
244
245     //printf("insert_source=[%s\n",insert_source.c_str());
246     db->query((char*)insert_source.c_str());
247     //db->query((char*)"INSERT INTO source VALUES(5,LES_len,path );");
248
249
250   point orig(0,0,0);
251   db->begin();
252     for(int zz=0; zz < LES_len; zz++)
253     {
254       orig.x = 0;
255       for(int xx = 0; xx < LES_len;xx++)
256       {
257         orig.y=0;
258         for(int yy=0; yy <LES_len;yy++)
259         {
260           int ind = indx(LES_model_sz,xx,yy,zz);
261           double filtred_u = filter(orig,DNS_sz,f,u);
262           double filtred_v = filter(orig,DNS_sz,f,v);
263           double filtred_w = filter(orig,DNS_sz,f,w);
264           LES_cellNo++;
265
266           orig.y += fw;
267           stringstream sind;
268           sind<<ind;
269           stringstream sfiltred_u;
270           sfiltred_u<<filtred_u;
271           stringstream sfiltred_v;
272           sfiltred_v<<filtred_v;
273           stringstream sfiltred_w;
274           sfiltred_w<<filtred_w;
275
276           insert_source="INSERT INTO data VALUES(";
277           insert_source += sind.str();
278           insert_source += ch;
279           insert_source += sfiltred_u.str();
280           insert_source += ch;
281           insert_source += sfiltred_v.str();
282           insert_source += ch;
283           insert_source += sfiltred_w.str();
```

```
284             insert_source += en;
285            db->query((char*)insert_source.c_str());
286           }
287           db->commit();
288          orig.x+= fw;
289        }
290        orig.z += fw;
291        printf(" done level %d \n",zz);
292      }
293
294    db->end();
295
296
297
298     delete[] u;
299     delete[] v;
300     delete[] w;
301
302
303     db->close();
304     delete db;
305
306     cout << out_dir << " Has been successfully created" << endl;
307
308
309   }
```

Here is the call graph for this function:



double core::dfdx (int *ind*, order_t *o*, double * *u*)

Referenced by plugins::Smagorinsky::stress().

Here is the caller graph for this function:

double core::dfdy (int *ind*, order_t *o*, double * *u*)

double core::dfdz (int *ind*, order_t *o*, double * *u*)

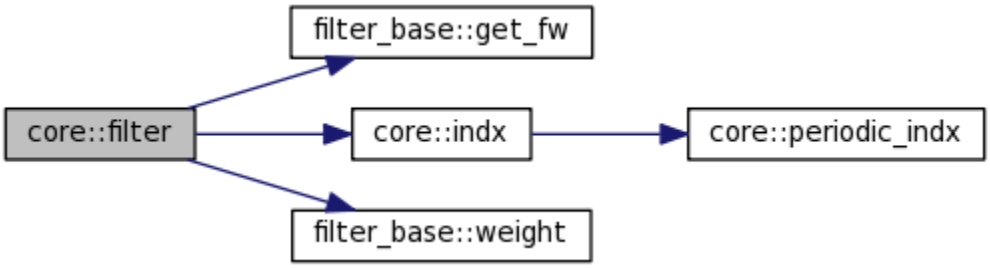
double core::filter (point & *pmin*, size *sz*, filter_base & *f*, double * *v*) `[private]`

References filter_base::get_fw(), indx(), filter_base::weight(), point::x, point::y, and point::z.

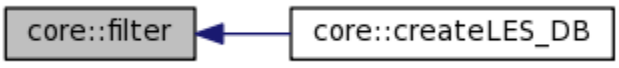
Referenced by createLES_DB().

```
312 {
313
314     int fw=f.get_fw();
315     double sum=0;
316
317     point pmax=pmin;
318     pmax.x+=fw;
319     pmax.y+=fw;
320     pmax.z+=fw;
321
322     for(int z =pmin.z; z < pmax.z; z++)
323      for(int x = pmin.x; x < pmax.x; x++)
324       for(int y = pmin.y; y < pmax.y; y++)
325       {
326         int ind = indx(sz,x,y,z); //sz here is DNS model size
327         //position in gaussian weight quibe
328         int xg = x-pmin.x;
329         int yg = y-pmin.y;
330         int zg = z-pmin.z;
331         //printf("Filter ind=%d xg=%d yg=%d zg=%d\n",ind,xg,yg,zg);
332         sum += v[ind]*f.weight(xg,yg,zg);
333
334       }
335
336    return sum;
337 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

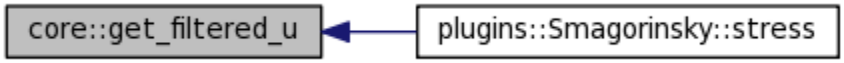


int core::get_filter_width ()

double * core::get_filtered_u ()

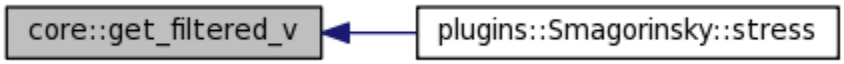
Referenced by plugins::Smagorinsky::stress().

Here is the caller graph for this function:



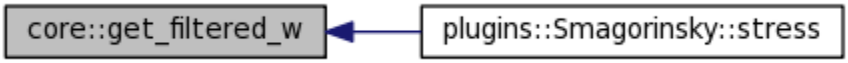
double * core::get_filtered_v ()

Here is the caller graph for this function:



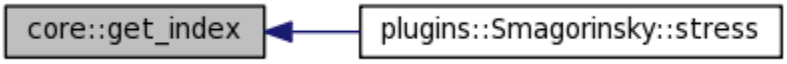
double * core::get_filtered_w ()

Here is the caller graph for this function:



int core::get_index (size & *pos*)

Here is the caller graph for this function:



int core::get_model_size () [inline]

References m_model_size.

int core::getdir (std::string *dir*, std::string *ext*, std::vector< std::string > & *vfiles*, const std::string & *optional* = "") [private]

References compare().

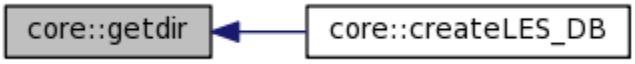
Referenced by createLES_DB().

```
408   {
409      DIR *dp;
410    struct dirent *dirp;
411    if((dp  = opendir(dir.c_str())) == NULL) {
412        cout << "Error(" << errno << ") opening " << dir << endl;
413        return errno;
414    }
415
416    list<string> lfiles;
417    const string empty;
418
419    while ((dirp = readdir(dp)) != NULL) {
420        string file = dirp->d_name;
421
422     //cout << "IG: file=" <<file << endl;
423     int idx = file.rfind('.');
424        if(idx != string::npos)
425        {
426          if(optional == empty)
427          {
428              if( file.substr(idx+1) == ext) lfiles.push_back(file);
429          }
430          else
431          {
432            int len = file.length();
433            int opt = optional.length();
434
435
436            if(len > opt)
437            {
438              string substr = file.substr(len-opt,opt);
439              if(substr == optional)  lfiles.push_back(file);
440            }
441
442          }
443       }
444    }
445    lfiles.sort(compare);
446    list<string>::iterator it;
447    for(it=lfiles.begin();it != lfiles.end(); it++)
448    {
449      vfiles.push_back(*it);
450    }
451
452    closedir(dp);
```

```
453    return vfiles.size();
454
455  }
```

Here is the call graph for this function:

Here is the caller graph for this function:

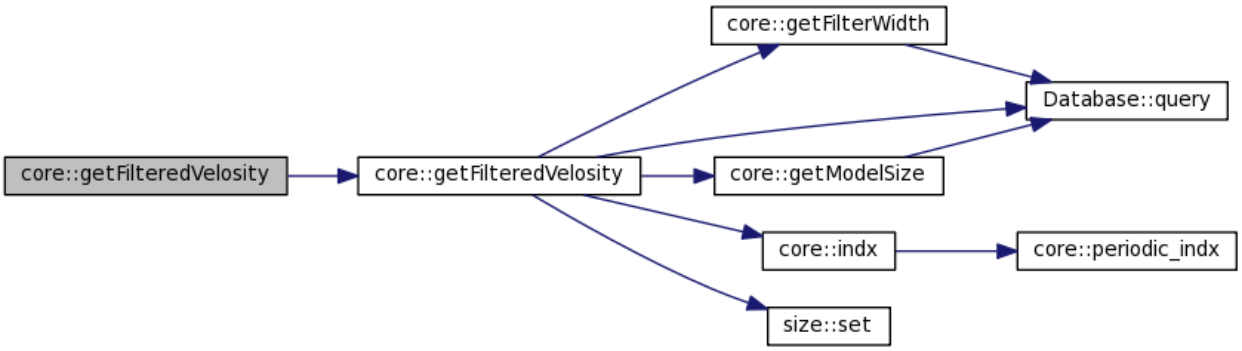


void core::getFilteredVelosity (LIST_POINTS & *pl*, LIST_DATA & *ld*, Database * *db*)

References getFilteredVelosity().

```
117  {
118    LIST_POINTS::iterator ip;
119
120     /*
121     list<int> tl;
122     list<int>::iterator it;
123     tl.push_back(5);
124     for(it = tl.begin() ; it != tl.end(); it++)
125     {
126     }
127     */
128
129    for(ip = pl.begin(); ip != pl.end(); ++ip)
130    {
131      point p=*ip;
132      FilteredData fd =  getFilteredVelosity(p,db);
133      dl.push_back(fd);
134    }
135  }
```

Here is the call graph for this function:



FilteredData core::getFilteredVelosity (point & *p*, Database * *db*)

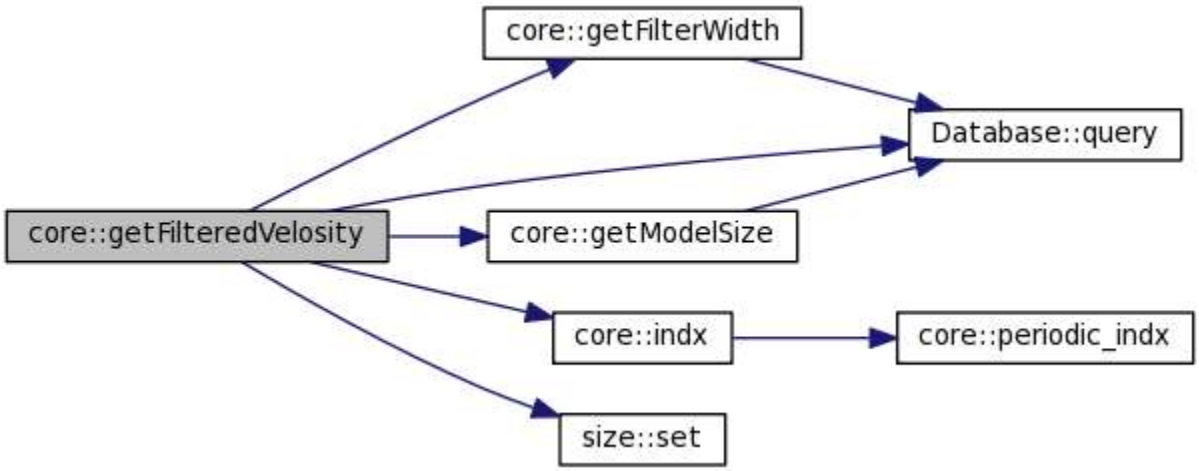
References getFilterWidth(), getModelSize(), indx(), m_db, m_fw, m_size, Database::query(), size::set(), point::x, point::y, and point::z.

Referenced by getFilteredVelosity(), and main().

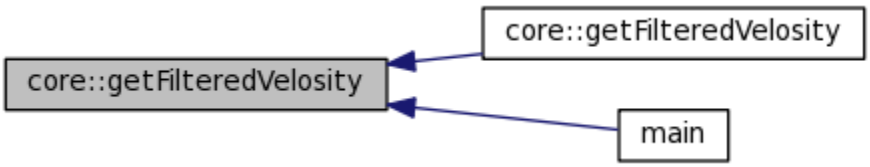
```
138   {
139     if(m_db != db)
140     {
141       m_db = db;
142       m_fw = getFilterWidth(db);
143       int len = getModelSize(db);
144       m_size.set(len,len,len);
145     }
146 
147     int ind = indx(m_size,p.x,p.y,p.z);
148     stringstream sind;
149     sind<<ind;
150 
151     string query="SELECT * FROM data WHERE  ind=";
152     query += sind.str();
153 
154     table res = db->query((char*)query.c_str());
155     table::iterator it;
156     for(it = res.begin(); it < res.end(); ++it)
157     {
158       row  rw = *it;
159       /*
160       cout << "Values: (ind=" << rw.at(0) <<
161                          ", u=" << rw.at(1) <<
162                          ", v=" << rw.at(2) <<
163                          ", w=" << rw.at(3) <<
164                          ")" << endl;
```

```
165         */
166         double u  = atof(rw.at(1).c_str());
167         double v  = atof(rw.at(2).c_str());
168         double w  = atof(rw.at(3).c_str());
169         FilteredData pnt(u,v,w);
170         return pnt;
171 
172       }
173     }
```

Here is the call graph for this function:



Here is the caller graph for this function:



int core::getFilterWidth (Database * *db*)

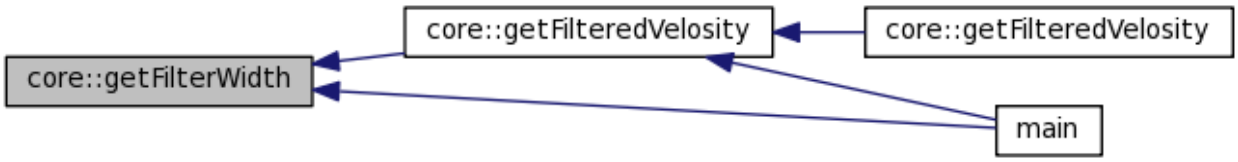
References Database::query().

Referenced by getFilteredVelosity(), and main().

```
104   {
105       table res = db->query((char*)"SELECT filter_width FROM source;");
106       table::iterator it =  res.begin();
107       return atoi((*it).at(0).c_str());
108   }
```

Here is the call graph for this function:

Here is the caller graph for this function:



int core::getModelSize (Database * *db*)

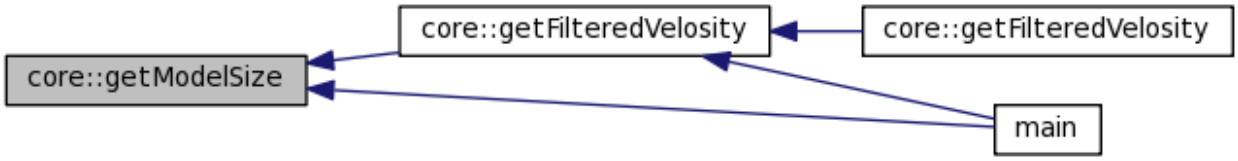
References Database::query().

Referenced by getFilteredVelosity(), and main().

```
110   {
111       table res = db->query((char*)"SELECT LES_model_sz FROM source;");
112       table::iterator it =  res.begin();
113       return atoi((*it).at(0).c_str());
114   }
```

Here is the call graph for this function:

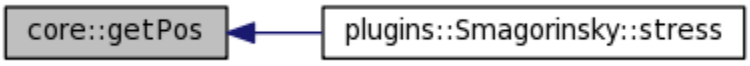
Here is the caller graph for this function:



size core::getPos ()

Referenced by plugins::Smagorinsky::stress().

Here is the caller graph for this function:



int core::indx (size & sz, int x, int y, int z) [private]

References size::lenX, size::lenY, size::lenZ, and periodic_indx().
Referenced by createLES_DB(), filter(), getFilteredVelosity(), and read_file().

```
511 {
512
513   int xp = periodic_indx(sz.lenX,x);
514   int yp = periodic_indx(sz.lenY,y);
515   int zp = periodic_indx(sz.lenZ,z);
516   int ind =  yp + sz.lenY*(xp + sz.lenX*zp);
517   return ind;
518 }
```

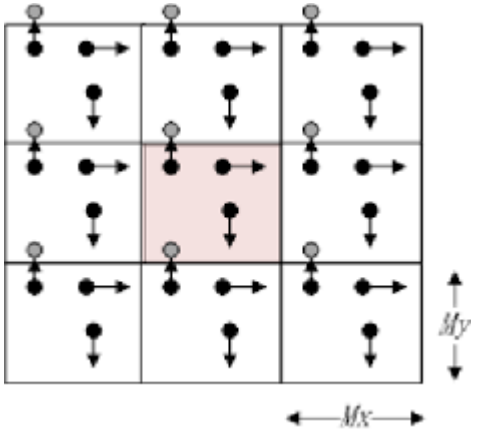


Figure 6.2 The code above implements the idea of periodic boundary conditions, where we are simulating infinity by a finite number of cells. If a point crosses the boundary, another one comes inside from the other side.

Here is the call graph for this function:

Here is the caller graph for this function:

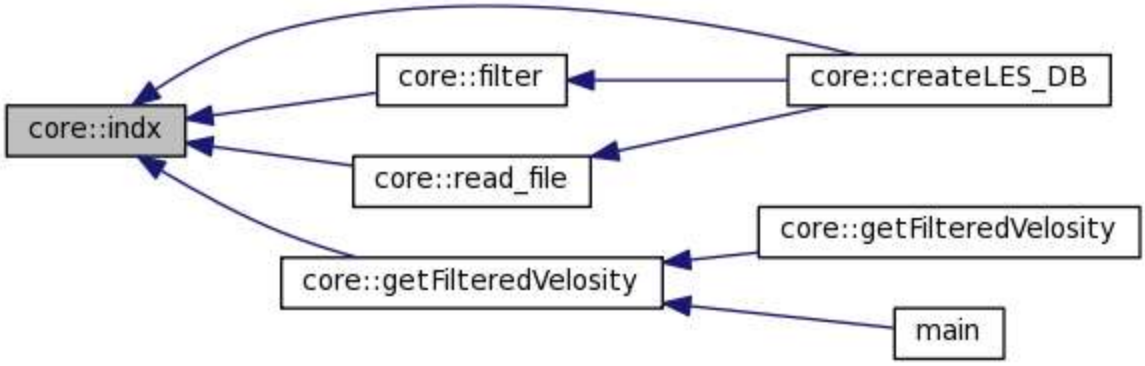


Figure 6.3 Diagram to show objects to references index class

void core::load_plugin ()

References construct(), m_handle, and m_plugin.

Referenced by model_stress().

```
46  {//this method dynamicly load plugin and granted access to core object
47
48     //void *handle = NULL;
49     if(!(m_handle = dlopen("lib/libplugin.so", RTLD_LAZY)))
50     {
51       std::cerr << "Plugin: " << dlerror() << std::endl;
52       return;
53     }
54     dlerror();
55
56    pluginConstructor construct = (plugins::Plugin* (*)(void)) dlsym(m_handle,
"construct");
57    char *error = NULL;
58    if((error = dlerror()))
59    {
60      std::cerr << "Plugin: " << dlerror() << std::endl;
61      dlclose(m_handle);
62      return;
63    }
64
65     //plugins::Plugin *plugin = construct();
66     //std::cout <<  plugin->toString() << std::endl;
67     //delete plugin;
68      m_plugin = construct();
69     //dlclose(handle);
70  }
```

Here is the call graph for this function:

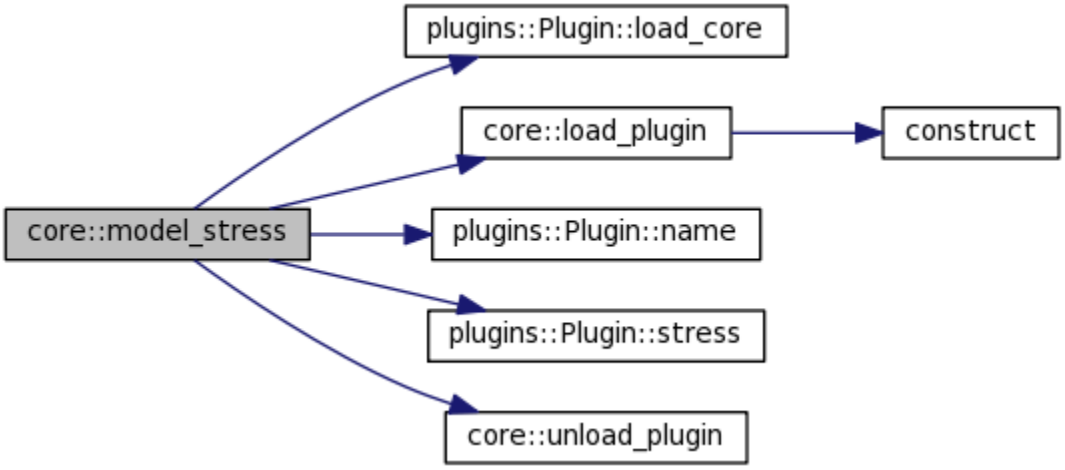
Here is the caller graph for this function:

void core::model_stress ()

References plugins::Plugin::load_core(), load_plugin(), m_plugin, plugins::Plugin::name(), plugins::Plugin::stress(), and unload_plugin().

```
340   {
341     load_plugin();
342     m_plugin->load_core(this );
343 
344     std::cout << "simulating stress by " << m_plugin->name() << " model" << std::endl;
345 
346     Matrix m(3,3);
347     m_plugin->stress(m);
348 
349     unload_plugin();
350 
351   }
```

Here is the call graph for this function:

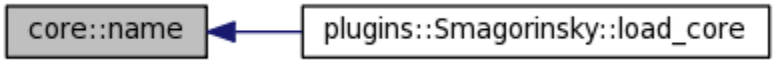


std::string core::name ()

Referenced by plugins::Smagorinsky::load_core().

```
86   {
87     return std::string(" The  core");
88   }
```

Here is the caller graph for this function:

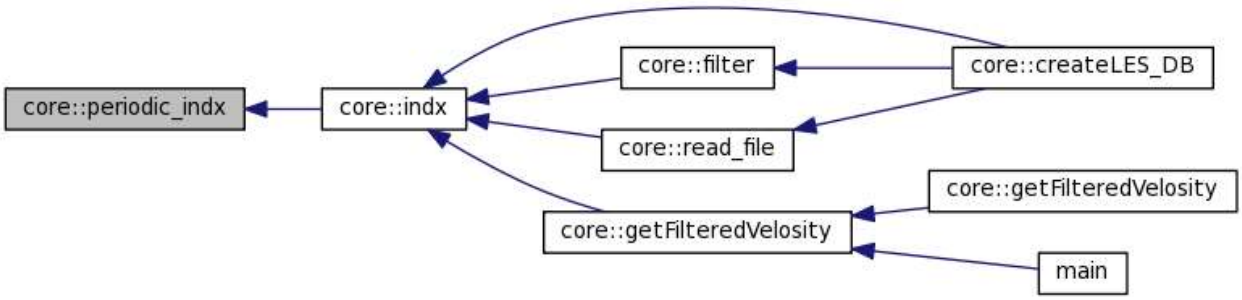


int core::periodic_indx (int size, int ind) [private]

Referenced by indx().

```
499 {
500   int res;
501   if ( ind < size  &&  ind >= 0) return ind;
502   if ( ind >= size ) { res = ind +1 - size; return res; }
503   if ( ind < 0 )  { res =  ind -1 + size; return res;}
504
505   // should not ever come to this point of return
506   // return unchange indx;
507    return ind;
508 }
```

Here is the caller graph for this function:



int core::read_file (string & *path*, string & *fname*, size & *sz*, int *iz*, double * *u*, double * *v*, double * *w*) `[private]`

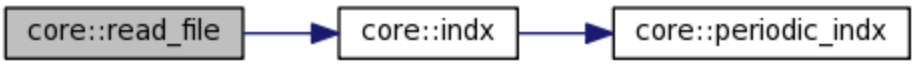
References indx(), and size::lenX.

Referenced by createLES_DB().

```
458 {
459       string fullname=path+fname;
```

```
460       ifstream infile(fullname.c_str());
461       string line;
462
463
464       int ix=0;
465       int iy=0;
466       int ind;
467       while (getline(infile, line))
468       {
469         //ind = ix + sz.x*(iy + sz.y*iz);
470         //          //ind = iy + sz.y*(ix + sz.x*iz);
471         //
472          ind = indx(sz,ix,iy,iz);
473
474         istringstream iss(line);
475         double v1,v2,v3;
476         if (!(iss >> v1 >> v2 >> v3))
477         {
478             break;
479         }
480
481         u[ind] = v1;
482         v[ind] = v2;
483         w[ind] = v3;
484
485
486         if(ix < sz.lenX) ix++;
487         if(ix == sz.lenX)
488         {
489           ix = 0;
490           iy += 1;
491         }
492
493
494       }
495   infile.close();
496 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

void core::set_size (size & *s*) [inline]

References m_size.

```
30 {m_size = s;}
```

void core::unload_plugin ()

References m_handle, and m_plugin.

Referenced by model_stress().

```
80   {
81     delete m_plugin;
82     dlclose(m_handle);
83   }
```

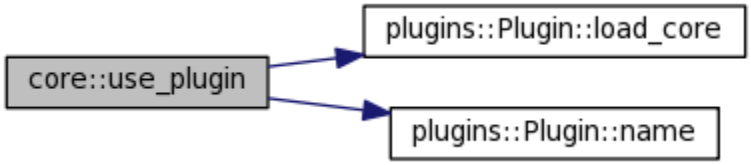
Here is the caller graph for this function:

void core::use_plugin ()

References plugins::Plugin::load_core(), m_plugin, and plugins::Plugin::name().

```
73   {
74     std::cout << "using plugin " <<std::endl;
75     std::cout << m_plugin->name() << std::endl;
76     m_plugin->load_core(this );
77   }
```

Here is the call graph for this function:



The object of the core system can be created by the following call.

6.2.6 MEMBER DATA

Database* core::m_db [private]

Referenced by getFilteredVelosity().

int core::m_fw [private]

Referenced by getFilteredVelosity().

void* core::m_handle [private]

Referenced by load_plugin(), and unload_plugin().

int core::m_model_size [private]

Referenced by get_model_size().

plugins::Plugin* core::m_plugin [private]

Referenced by load_plugin(), model_stress(), unload_plugin(), and use_plugin().

size core::m_size [private]

core *a = new(core);

The core class is a singleton, so only one instance of this object will be available. After the object of the core has been instantiated the user can access its public members listed below:

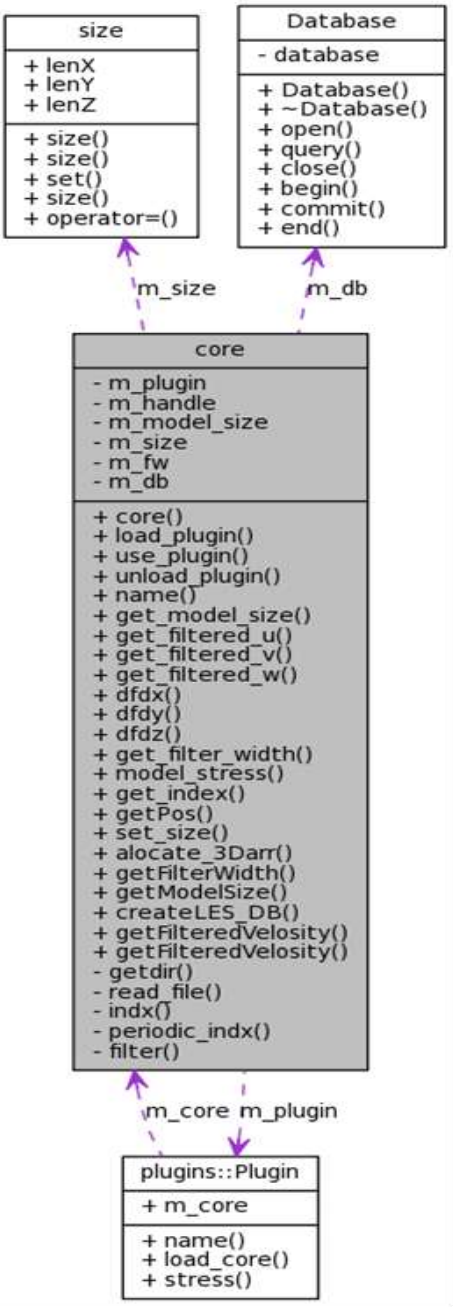


Figure 6.4 The relationships between the core class and the database class is the call graph that demonstrates the hierarchy employed by the core to generate a LES database. For example, the core applies filters of a given width and weight to those elements

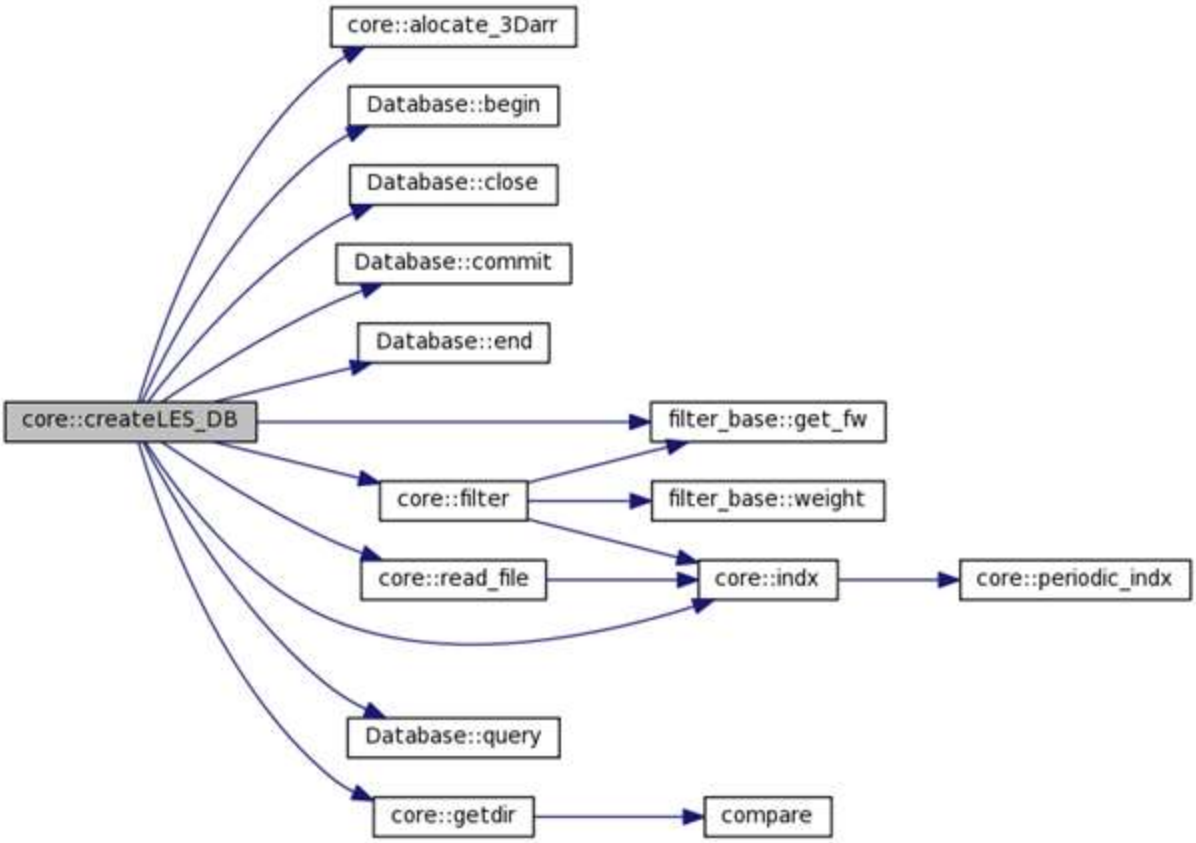


Figure 6.5 The hierarchy of instructions issued by the core to generate an LES database and comparator. This method is referred in the **Figure** 6.4

The core is aware of the periodicity of the data and compensates for this when reading beyond the file width. It is capable of identifying the data to be manipulated, carrying out the requested filtering and comparing the LES and DNS models on a time scale of about 10^{-3} seconds.

6.3 FILTERING OPERATIONS

6.3.1 FILTER_BASE CLASS REFERENCE

#include <filter.h>

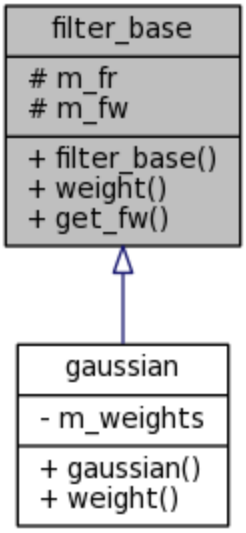


Figure 6.6 Inheritance diagram for filter_base

Public Member Functions

- filter_base (int fr)
- virtual double weight (int x, int y, int z)=0
- int get_fw ()

Protected Attributes

- int m_fr
- int m_fw

6.3.2 CONSTRUCTOR & DESTRUCTOR

filter_base::filter_base (int *fr*) [inline]

References m_fw.

```
7 :m_fr(fr){m_fw = 2*fr+1;};
```

6.3.3 MEMBER FUNCTION DOCUMENTATION

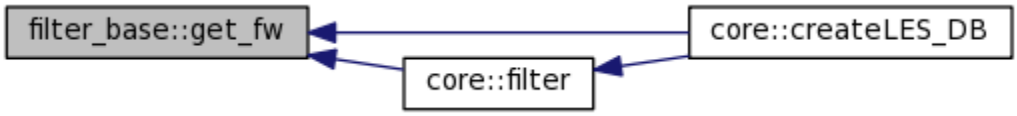
int filter_base::get_fw () [inline]

References m_fr.

Referenced by core::createLES_DB(), and core::filter().

```
9 {return 2*m_fr+1;}
```

Here is the caller graph for this function:

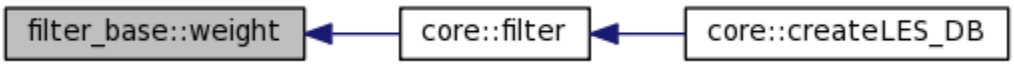


virtual double filter_base::weight (int *x*, int *y*, int *z*) [pure virtual]

Implemented in Gaussian.

Referenced by core::filter().

Here is the caller graph for this function:



6.3.4 MEMBER DATA

6.3.4.1 int filter_base::m_fr [protected]

Referenced by gaussian::gaussian(), and get_fw().

int filter_base::m_fw [protected]

Referenced by filter_base(), and gaussian::gaussian().

6.4 FILTEREDDATA CLASS REFERENCE

#include <point.h>

6.4.1 PUBLIC MEMBER FUNCTIONS

FilteredData ()

FilteredData (double u, double v, double w)

FilteredData & operator= (const FilteredData &pp)

FilteredData (const FilteredData &pp)

6.4.2 PUBLIC ATTRIBUTES

- POINT3D p

6.4.3 CONSTRUCTOR & DESTRUCTOR

FilteredData::FilteredData () [inline]

```
11 {//Default constructor of Filterdata class};
```

FilteredData::FilteredData (double *u*, double *v*, double *w*) [inline]

References p.

```
13  { //Constructor of Filter Data class
14    p[0] = u;
15    p[1] = v;
16    p[2] = w;
17
18  };
```

FilteredData::FilteredData (const FilteredData & *pp*) [inline]

References p.

```
29  { //Copy constructor of FilterData class
30    p[0] = pp.p[0];
31    p[1] = pp.p[1];
32    p[2] = pp.p[2];
33  };
```

6.4.4 MEMBER FUNCTION

FilteredData& FilteredData::operator= (const FilteredData & *pp*) [inline]

References p.

```
21  { // Method to define operator equal which is  going to be used in comparison operator
22     p[0] = pp.p[0];
23     p[1] = pp.p[1];
24     p[2] = pp.p[2];
25  };
```

6.4.5 MEMBER DATA

6.4.5.1 POINT3D FilteredData::p

Referenced by FilteredData(), main(), and operator=().

6.5 GAUSSIAN CLASS REFERENCE

#include <gaussian.h>

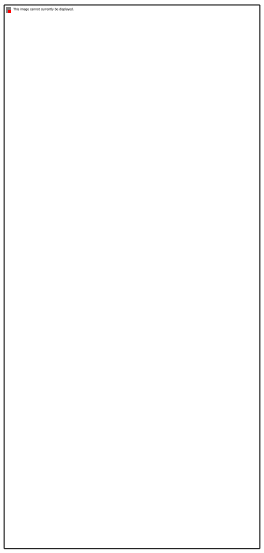
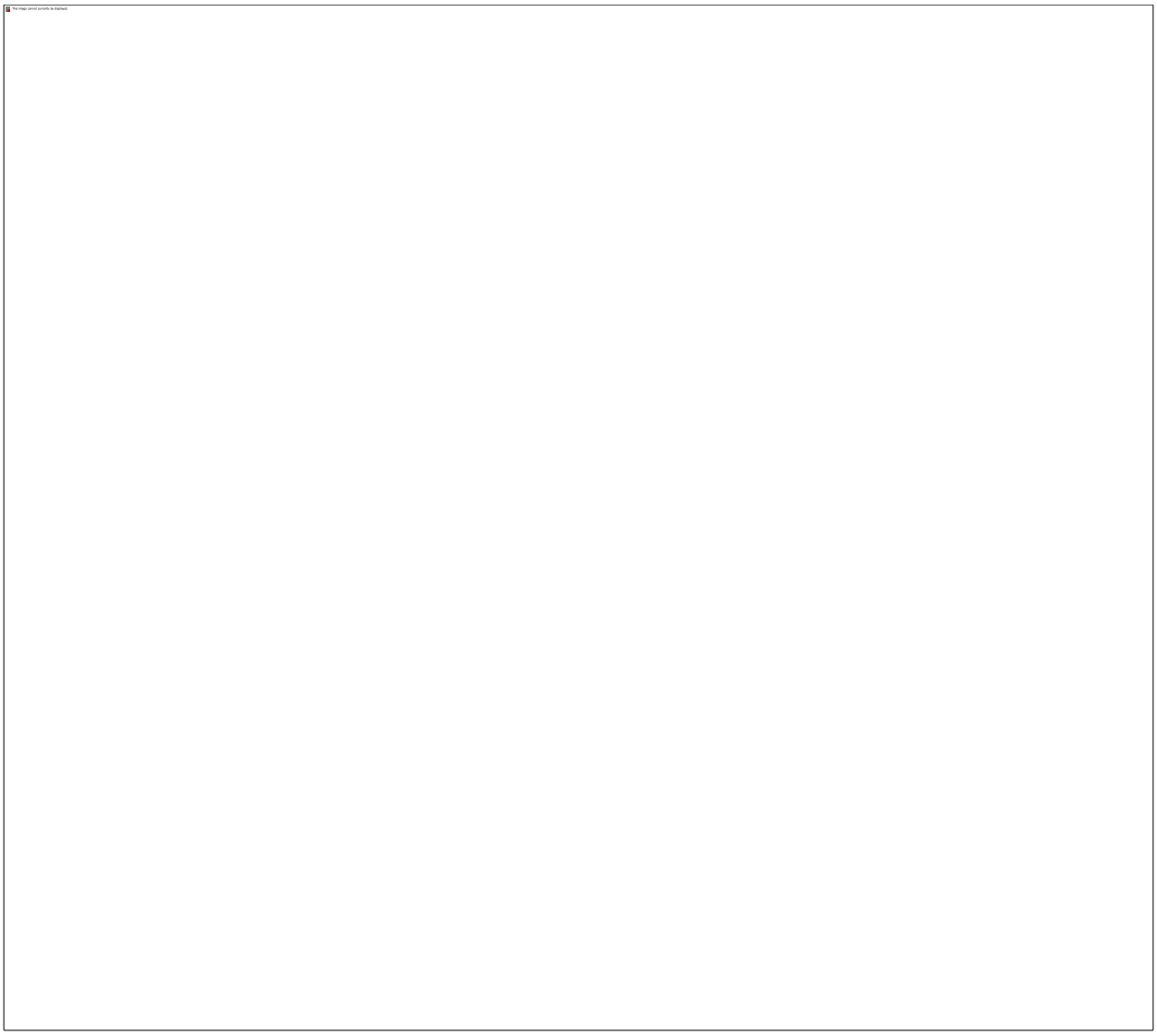


Figure 6.7 Inheritance diagram for gaussian

Collaboration diagram for Gaussian:



6.5.1 PUBLIC MEMBER FUNCTIONS

- gaussian (int fr)
- virtual double weight (int x, int y, int z)

6.5.2 PRIVATE ATTRIBUTES

- array3D m_weights

6.5.3 CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

gaussian::gaussian (int *fr*)

References filter_base::m_fr, filter_base::m_fw, and m_weights.

```
6                       :filter_base(fr)
7 {
8   int sp = m_fw*m_fw;
9   printf("m_fw=%d m_fr=%d\n",m_fw,m_fr);
10
11
12    m_weights.resize(m_fw);
13    double sum = 0;
14    for(int z = -m_fr; z <= m_fr; z++)
15    {
16      m_weights[z+m_fr].resize(m_fw);
17      for(int x = -m_fr; x <= m_fr; x++)
18      {
19        m_weights[z+m_fr][x+m_fr].resize(m_fw);
20        for(int y = -m_fr; y <= m_fr; y++)
21        {
22          double w = exp(-6*(x*x + y*y + z*z)/sp);
23          sum += w;
24          m_weights[z+m_fr][x+m_fr][y+m_fr] =  w ;
25        }
26      }
27    }
28
29
30    for(int z = -m_fr; z <= m_fr; z++)
31    {
32      for(int x = -m_fr; x <= m_fr; x++)
33      {
34        for(int y = -m_fr; y <= m_fr; y++)
35        {
36          m_weights[z+m_fr][x+m_fr][y+m_fr] /= sum;
37        }
38      }
39    }
40
41 // check weights
42 /*
43    sum = 0;
44    for(int z = -m_fr; z <= m_fr; z++)
45      for(int x = -m_fr; x <= m_fr; x++)
```

```
46         for(int y = -m_fr; y <= m_fr; y++)
47           sum += m_weights[z+m_fr][x+m_fr][y+m_fr];
48
49 printf("done gaussian sum = %f\n",sum);
50 */
51
52 }
```

6.5.4 MEMBER FUNCTION

double gaussian::weight (int *x*, int *y*, int *z*) [virtual]

Implements filter_base.

References m_weights.

```
55 {
56  return m_weights[z][x][y];
57 }
```

6.5.5 MEMBER DATA

array3D gaussian::m_weights [private]

Referenced by gaussian(), and weight().

7 SMAGORINSKY PLUGIN

7.1 INTRODUCTION

To begin using the platform and evaluate potential LES models, the user has to implement a small piece of code for the subgrid model. Because it is based on a plugin architecture, the language that is used is not particularly important. One of the first sub-grid-scale models was suggested in Smagorinsky's classic work (Smagorinsky, 1963), and it can be summarised as:

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = -2\mu_{sgs}\bar{S}_{ij} = -2(C_s\Delta)^2 \mid \bar{S} \mid \bar{S}_{ij} \tag{7.1}$$

where

$$\bar{S}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right), \qquad \mid \bar{S} \mid = \sqrt{2\bar{S}_{ij}\bar{S}_{ij}} \tag{7.2}$$

In (7.1), the eddy viscosity is modelled by

$$\mu_{sgs} = \rho(C_s, \Delta)^2 \mid \bar{S} \mid, \qquad \Delta = (volume)^{\frac{1}{3}}, \quad C_s = 0.11 \tag{7.3}$$

Let us see what methods and data are required to implement a plugin to evaluate this relatively simple model. First, it is necessary to access a filtered velocity, and then we require the ability to calculate its derivatives. Furthermore, the plugin will need to know the geometrical position to be interrogated and the size of the model. The core system will supply these data. The plugin will return two variables – stress and eddy viscosity. As we have mentioned above, this is a fully independent component and after compilation will reside in the plugin.so share library.

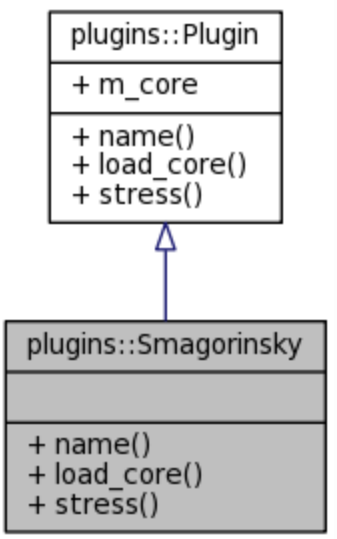


Figure 7.1 illustrates schematically how the user can implement a plugin to form a LES database from DNS data and evaluate the accuracy of, in this case, a Smagorinsky model. However, it should be noted that the user is free to devise any LES model and check its efficacy.

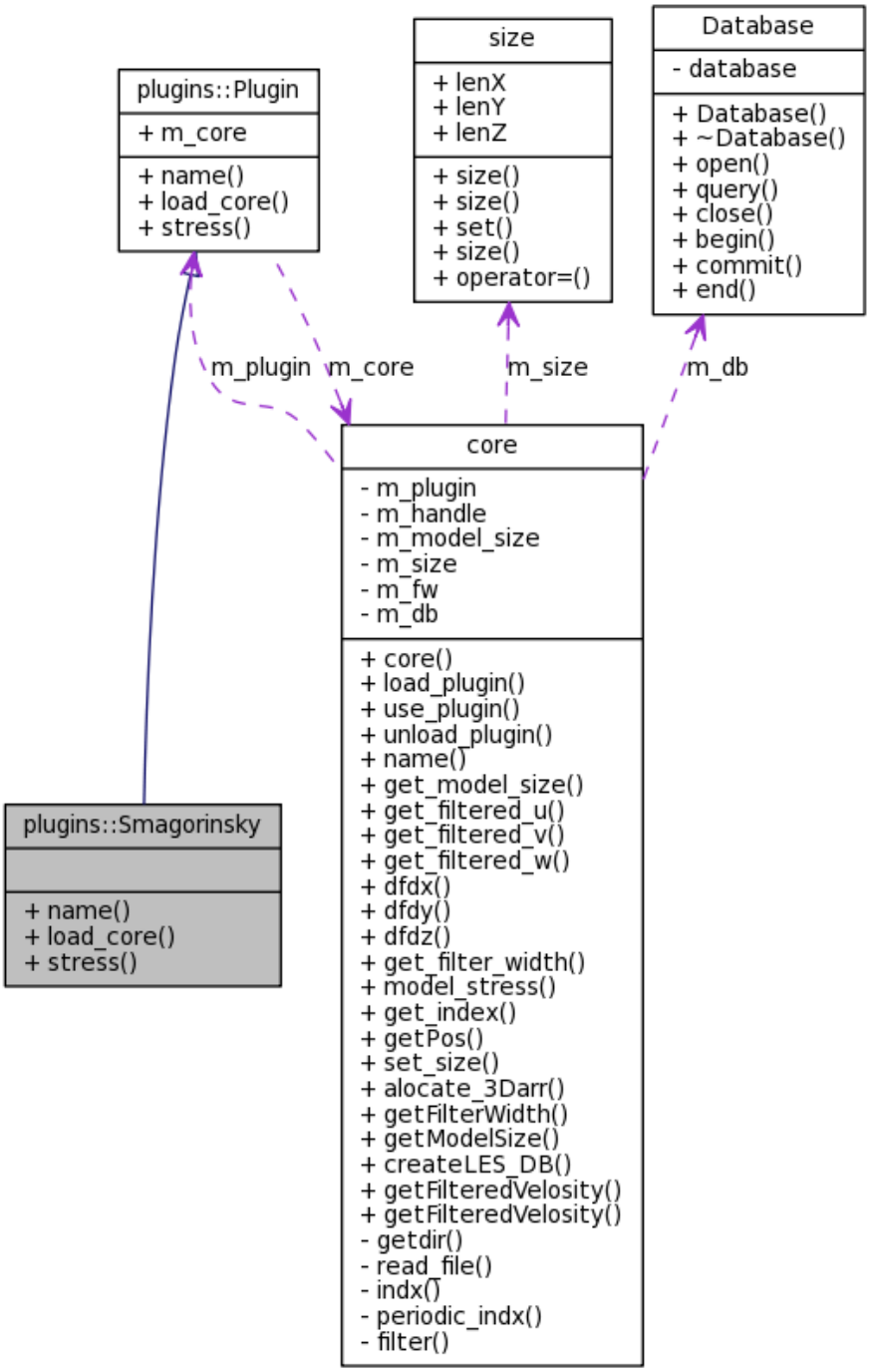


Figure 7.2 The diagram shows the interaction between the plugin and the core.

The core subsumes ‘know-how’ on locating the DNS data to be analysed, methods of filtering, creating a LES database and comparing DNS and LES data. The operations of the core are invisible to the CFD developer.

Our platform has a built-in directory of examples that includes a subdirectory in which plugin examples are provided. It includes a template which can be used to develop and test alternative LES models

7.2 DETAILED PLUGIN DOCUMENTATION

plugins:: Plugin Class Reference

#include <plugin.hpp>

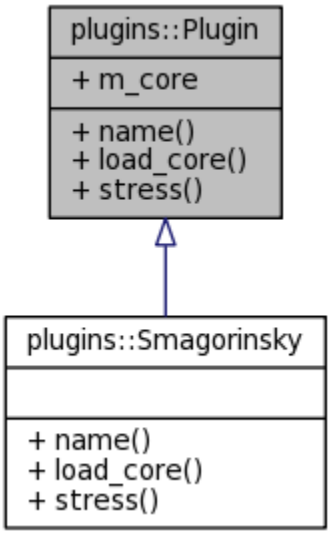


Figure 7.3 Inheritance diagram for plugins::Plugin

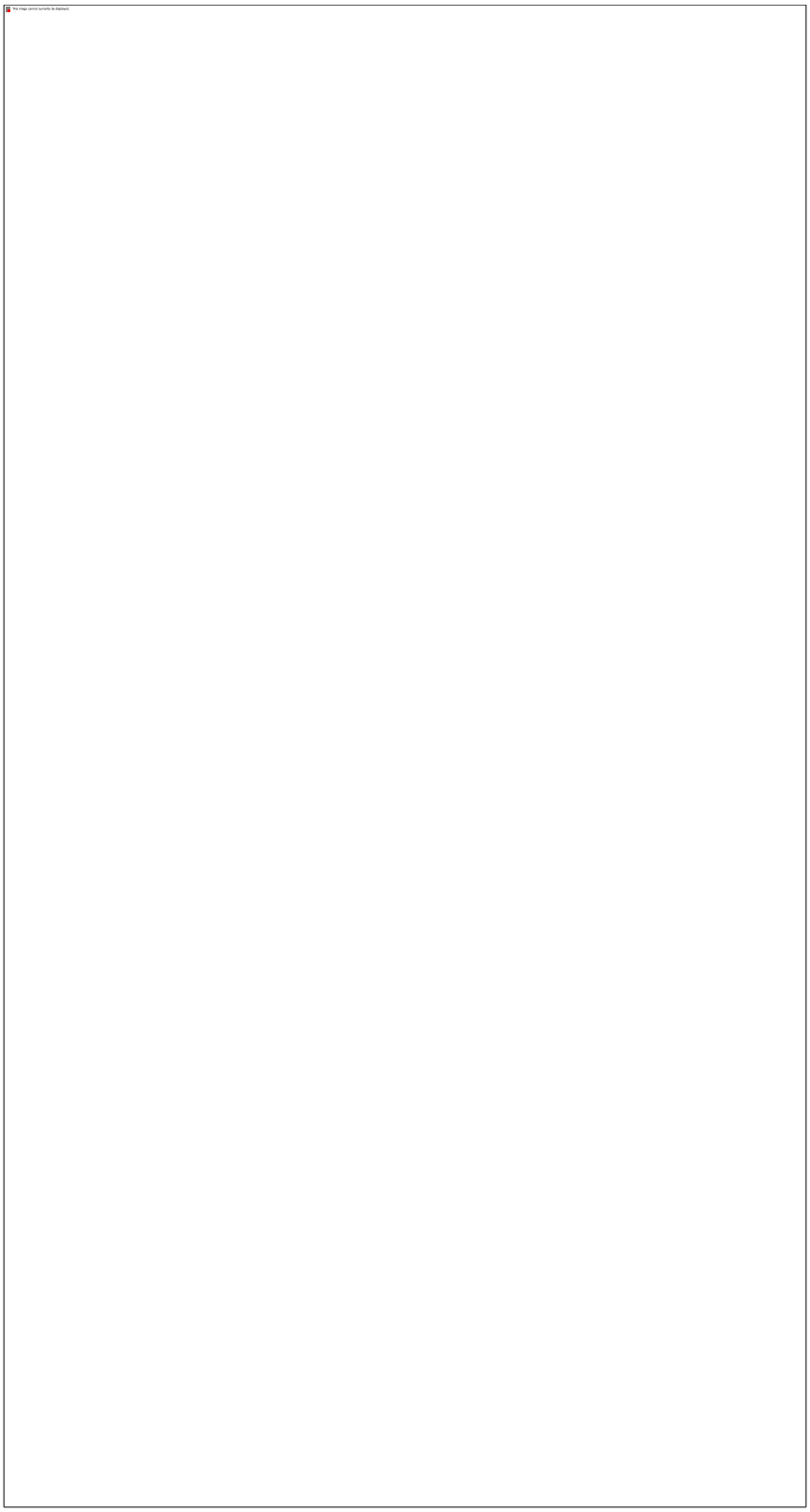


Figure 7.4 Collaboration diagram for plugins::Plugin:

7.2.1 PUBLIC MEMBER FUNCTIONS

- virtual std::string name ()=0
- virtual void load_core (core *a)=0
- virtual void stress (Matrix &m)=0

7.2.2 PUBLIC ATTRIBUTES

- core * m_core

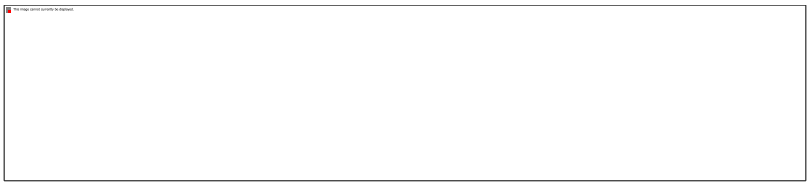
7.2.3 MEMBER FUNCTION DOCUMENTATION

virtual void plugins::Plugin::load_core (core * *a*) [pure virtual]

Implemented in plugins:: Smagorinsky.

Referenced by core::model_stress(), and core::use_plugin().

Here is the caller graph for this function:

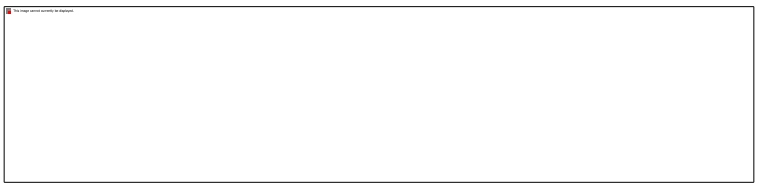


virtual std::string plugins::Plugin::name () [pure virtual]

Implemented in plugins:: Smagorinsky.

Referenced by core::model_stress(), and core::use_plugin().

Here is the caller graph for this function:



virtual void plugins::Plugin::stress (Matrix & *m*) [pure virtual]

Implemented in plugins:: Smagorinsky.

Referenced by core::model_stress().

Here is the caller graph for this function:

void plugins::Smagorinsky::stress (Matrix & *s*) [inline, virtual]

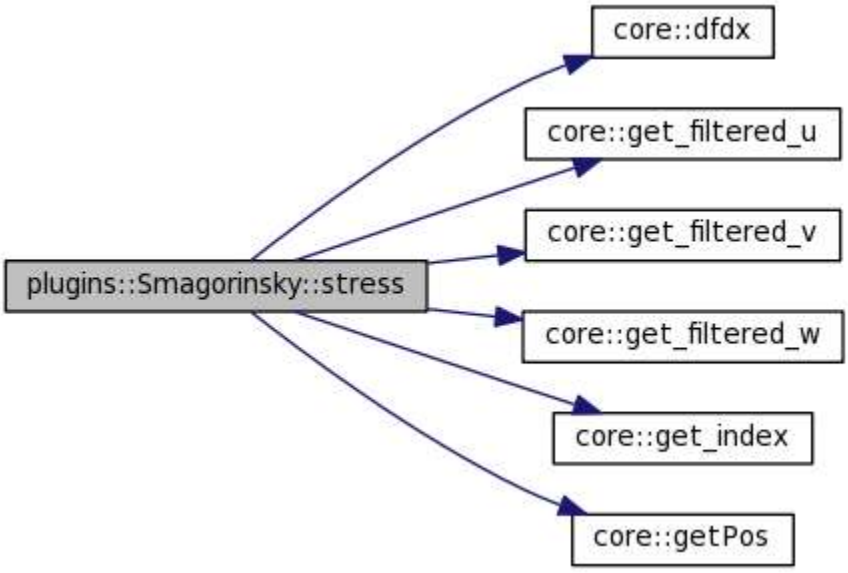
Implements plugins:: Plugin.

References core::dfdx(), FOUR, core::get_filtered_u(), core::get_filtered_v(), core::get_filtered_w(), core::get_index(), core::getPos(), and plugins::Plugin::m_core.

```
24   {
25     double *u = m_core->get_filtered_u();
26     double *v = m_core->get_filtered_v();
27     double *w = m_core->get_filtered_w();
28     size pos = m_core->getPos();
29     int ind = m_core->get_index(pos);
30 
31     order_t order=FOUR;
32 
33 /*
34     double dudx = m_core->dfdx(ind,order,u);
35     double dudy = m_core->dfdy(ind,order,u);
36     double dudz = m_core->dfdz(ind,order,u);
37 
38 
39     double dvdx = m_core->dfdx(ind,order,v);
40     double dvdy = m_core->dfdy(ind,order,v);
41     double dvdz = m_core->dfdz(ind,order,v);
42 
43     double dwdx = m_core->dfdx(ind,order,w);
44     double dwdy = m_core->dfdy(ind,order,w);
45     double dwdz = m_core->dfdz(ind,order,w);
46 */
47     double dudx = m_core->dfdx(ind,order,u);
48     double dudy = 0;
49     double dudz = 0;
50 
51 
52     double dvdx = 0;
53     double dvdy = 0;
54     double dvdz = 0;
55 
56     double dwdx = 0;
57     double dwdy = 0;
58     double dwdz = 0;
59 
60 
```

```
61
62
63      s(0,0) = -0.5*( dudx + dudx );
64      s(0,1) = -0.5*( dudy + dvdx );
65      s(0,2) = -0.5*( dudz + dwdx );
66      s(1,1) = -0.5*( dvdy + dvdy );
67      s(1,2) = -0.5*( dvdz + dwdx );
68      s(2,2) = -0.5*( dwdz + dwdz );
69
70      s(1,0) = s(0,1);
71      s(2,0) = s(0,2);
72      s(2,1) = s(1,2);
73
74
75    }
```

Here is the call graph for this function:



7.2.4 MEMBER DATA DOCUMENTATION

core* plugins::Plugin::m_core

Referenced by plugins::Smagorinsky::load_core(), and plugins::Smagorinsky::stress().

7.3 POINT OBJECTS

point Class Reference

#include <point.h>

7.3.1 PUBLIC MEMBER FUNCTIONS

- point ()
- point (int init_x, int init_y, int init_z)
- point (const point &s)
- point & operator= (const point &s)
- bool operator< (const point &rhs) const

7.3.2 PUBLIC ATTRIBUTES

- int x
- int y
- int z

7.3.3 CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

point::point () [inline]

```
41 {};
```

point::point (int *init_x*, int *init_y*, int *init_z*) [inline]

References x, y, and z.

```
43   {
44     x = init_x;
45     y = init_y;
46     z = init_z;
47   }
```

point::point (const point & *s*) [inline]

```
49 : x(s.x),y(s.y),z(s.z){};
```

7.3.4 MEMBER FUNCTION

bool point::operator< (const point & *rhs*) const [inline]

References x, y, and z.

```
62    {
63       if( (x*x + y*y + z*z) < (rhs.x*rhs.x + rhs.y*rhs.y + rhs.z*rhs.z) ) return true;
64       else return false;
65    }
```

point& point::operator= (const point & *s*) [inline]

References x, y, and z.

```
52    {
53       x = s.x;
54       y = s.y;
55       z = s.z;
56       return *this;
57    }
```

7.3.5 MEMBER DATA

- **int point::x**

Referenced by core::createLES_DB(), core::filter(), core::getFilteredVelosity(), operator<(), operator=(), and point().

- **int point::y**

Referenced by core::createLES_DB(), core::filter(), core::getFilteredVelosity(), operator<(), operator=(), and point().

- **int point::z**

Referenced by core::createLES_DB(), core::filter(), core::getFilteredVelosity(), operator<(), operator=(), and point().

7.4 OBJECTS OF TYPE SIZE

7.4.1 SIZE CLASS REFERENCE

#include <plugin.hpp>

7.4.2 PUBLIC MEMBER FUNCTIONS

- size ()
- size (int init_x, int init_y, int init_z)
- void set (int init_x, int init_y, int init_z)
- size (const size &s)
- size & operator= (const size &s)

7.4.3 PUBLIC ATTRIBUTES

- int lenX
- int lenY
- int lenZ

7.4.4 CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

size::size () [inline]

```
20 {};
```

size::size (int *init_x*, int *init_y*, int *init_z*) [inline]

References lenX, lenY, and lenZ.

```
22      {
23              lenX = init_x;
24              lenY = init_y;
25              lenZ = init_z;
26      }
```

size::size (const size & *s*) [inline]

```
34 : lenX(s.lenX),lenY(s.lenY),lenZ(s.lenZ){};
```

7.4.5 MEMBER FUNCTION

size& size::operator= (const size & *s*) [inline]

References lenX, lenY, and lenZ.

```
37      {
38              lenX = s.lenX;
39              lenY = s.lenY;
40              lenZ = s.lenZ;
41              return *this;
42  }
```

void size::set (int *init_x*, int *init_y*, int *init_z*) [inline]

References lenX, lenY, and lenZ.

Referenced by core::getFilteredVelosity().

```
28      {
29              lenX = init_x;
30              lenY = init_y;
31              lenZ = init_z;
32      }
```

Here is the caller graph for this function:

7.4.6 MEMBER DATA

int size::lenX

Referenced by core::alocate_3Darr(), core::indx(), operator=(), core::read_file(), set(), and size().

int size::lenY

Referenced by core::alocate_3Darr(), core::indx(), operator=(), set(), and size().

int size::lenZ

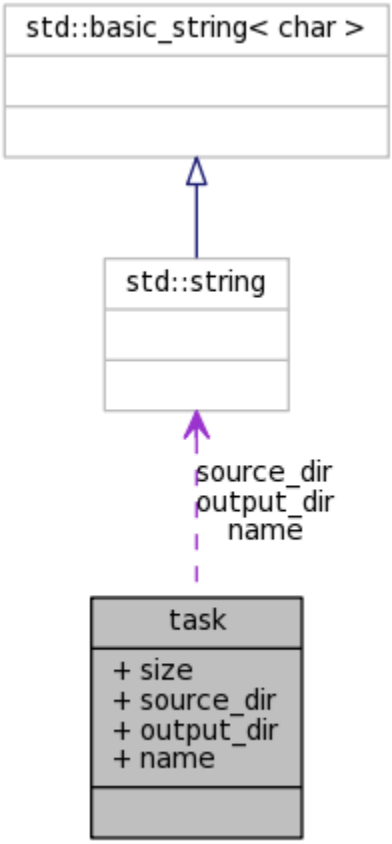
Referenced by core::alocate_3Darr(), core::indx(), operator=(), set(), and size().

7.4.7 FULL PLATFORM DETAILED PROJECT ORGANIZATION

7.5 TASK STRUCT REFERENCE

#include <task.h>

Collaboration diagram for task:



7.5.1 PUBLIC ATTRIBUTES

- int size
- string source_dir
- string output_dir
- string name

7.5.2 MEMBER DATA DOCUMENTATION

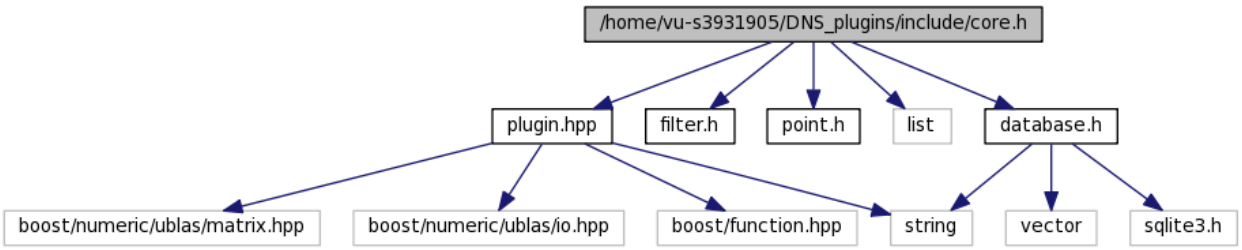
- **string task:: name**
- **string task::output_dir**
- **int task:: size**
- **string task::source_dir**

7.5.3 FILE DOCUMENTATION

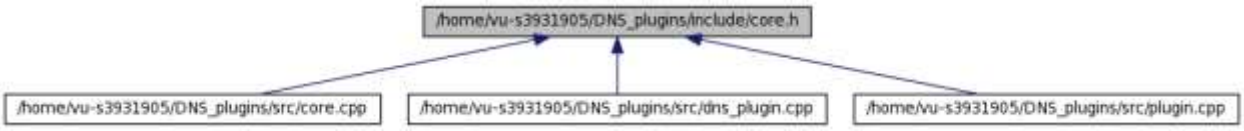
7.6 CORE.H FILE REFERENCE

```
#include "plugin.hpp"
#include "filter.h"
#include "point.h"
#include <list>
#include "database.h"
```

Include dependency graph for core.h:



This graph shows which files directly or indirectly include this file:



7.6.1 CLASSES

- class core

7.6.2 TYPEDEFS

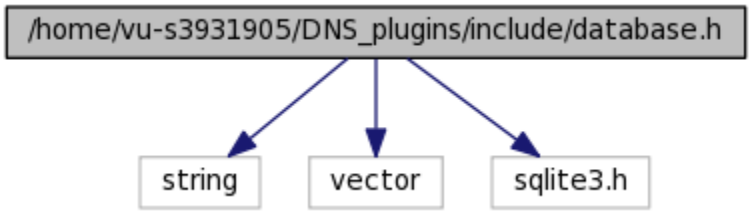
- typedef list< point > LIST_POINTS
- typedef list< FilteredData > LIST_DATA

7.6.3 TYPEDEF DOCUMENTATION

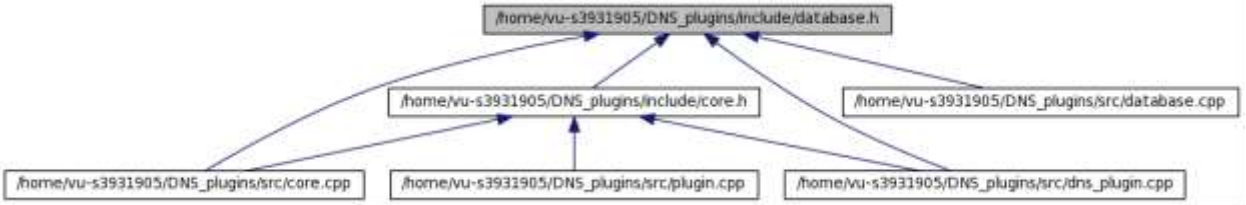
- **typedef list<FilteredData> LIST_DATA**
- **typedef list<point> LIST_POINTS**

7.7 DATABASE.H FILE REFERENCE

#include <string>
#include <vector>
#include <sqlite3.h>
Include dependency graph for database.h:



This graph shows which files directly or indirectly include this file:



7.7.1 CLASSES

- class Database

7.7.2 TYPEDEFS

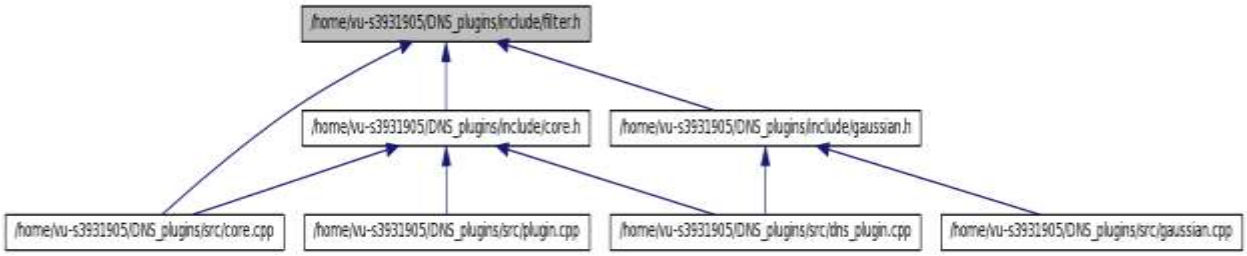
- typedef std::vector< std::string > row
- typedef std::vector< row > table

7.7.3 TYPEDEF DOCUMENTATION

- **typedef std::vector<std::string> row**
- **typedef std::vector<row> table**

7.8 FILTER.H FILE REFERENCE

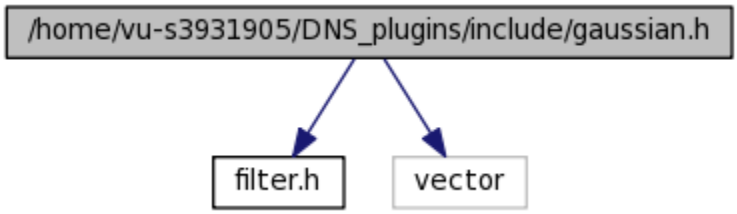
This graph shows which files directly or indirectly include this file:



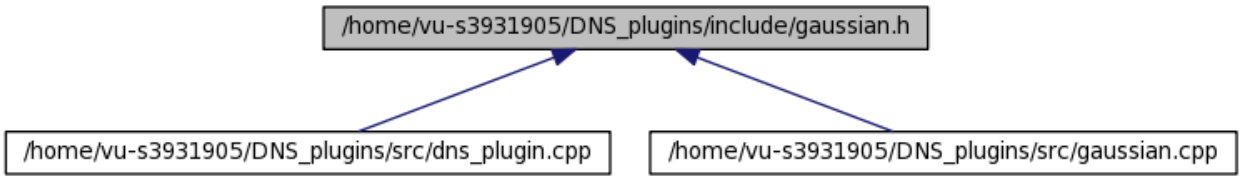
7.8.1 CLASSES

- class filter_baseGAUSSIAN.H FILE REFERENCE

#include "filter.h"
#include <vector>
Include dependency graph for gaussian.h:



This graph shows which files directly or indirectly include this file:



7.8.2 CLASSES

- class gaussian

7.8.3 TYPEDEFS

- typedef std::vector< std::vector< std::vector< double > > > array3D

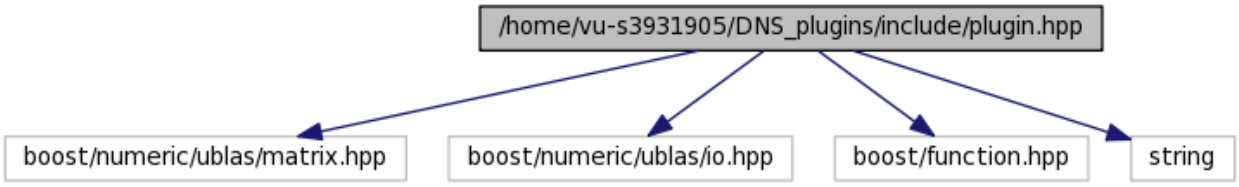
7.8.4 TYPEDEF DOCUMENTATION

typedef std::vector<std::vector<std::vector<double> > > array3D

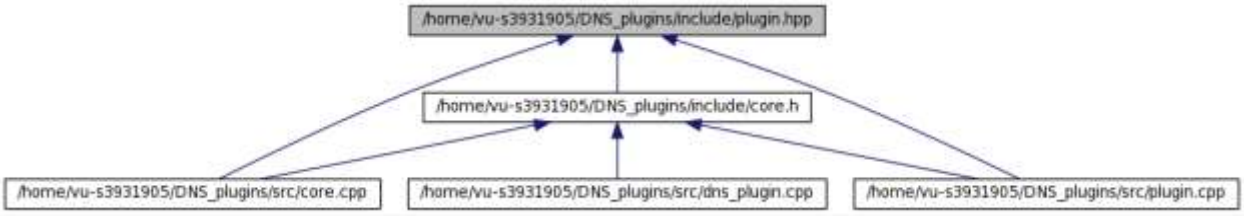
7.9 MAINPAGE.DOX FILE REFERENCE

7.10 PLUGIN.HPP FILE REFERENCE

#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/function.hpp>
#include <string>
Include dependency graph for plugin.hpp:



This graph shows which files directly or indirectly include this file:



7.10.1 CLASSES

- class size
- class plugins::Plugin

7.10.2 NAMESPACES

- namespace plugins

7.10.3 TYPEDEFS

- typedef matrix< double > Matrix

7.10.4 ENUMERATIONS

- enum order_t { ONE = 1, FOUR = 4 }

7.10.5 TYPEDEF DOCUMENTATION

typedef matrix<double> Matrix

7.10.6 ENUMERATION TYPE DOCUMENTATION

enum order_t

Enumerator:

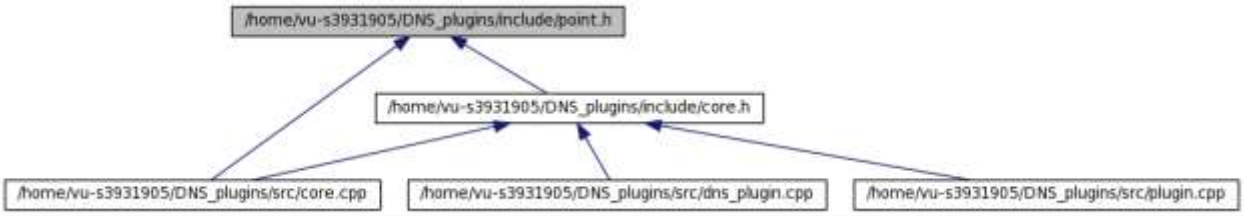
ONE

FOUR

16 { ONE=1,FOUR=4 } ;

7.11 POINT.H FILE REFERENCE

This graph shows which files directly or indirectly include this file:



7.11.1 Classes

- class FilteredData
- class point

7.11.2 Typedefs

- typedef double POINT3D [3]

7.11.3 TYPEDEF DOCUMENTATION

typedef double POINT3D[3]

7.12 TASK.H FILE REFERENCE

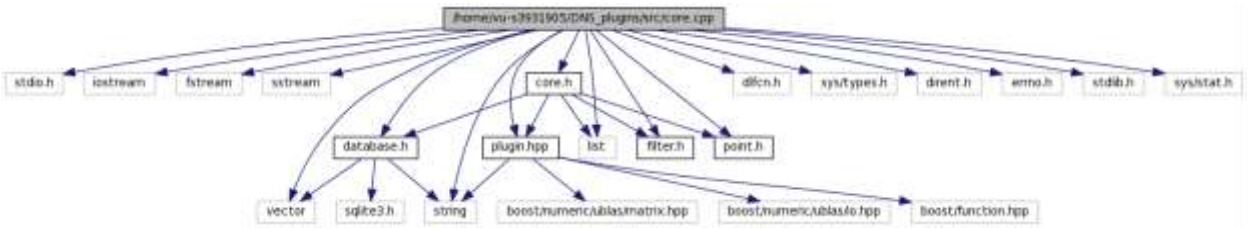
7.12.1 CLASSES

- struct task

7.13 CORE.CPP FILE REFERENCE

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <list>
#include <dlfcn.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdlib.h>
#include "database.h"
#include "plugin.hpp"
#include "core.h"
#include "filter.h"
#include "point.h"
#include <sys/stat.h>
```

Include dependency graph for core.cpp:



7.13.1 TYPEDEFS

- typedef boost::function< plugins::Plugin *()> pluginConstructor

7.13.2 FUNCTIONS

- bool compare (const std::string &first, const std::string &second)

7.13.3 TYPEDEF DOCUMENTATION

7.13.3.1 typedef boost::function<plugins::Plugin* ()> pluginConstructor

7.13.4 FUNCTION DOCUMENTATION

7.13.4.1 bool compare (const std::string & *first*, const std::string & *second*)

Referenced by core::getdir().

```
26 {
27
28         int pos1=first.find('_')+1;
29         int pos2=second.find('_')+1;
30         int one = atoi(first.substr(pos1,first.length()-8).c_str());
31         int two = atoi(second.substr(pos2,second.length()-8).c_str());
32
33         if(one > two) return false;
34         return true;
35
36 }
```

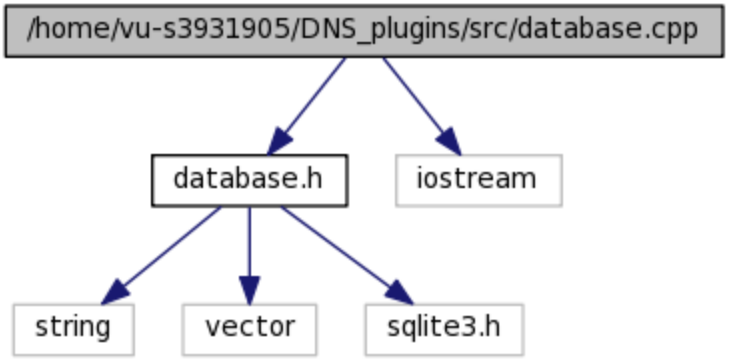
Here is the caller graph for this function:

7.14 DATABASE.CPP FILE REFERENCE

#include "database.h"

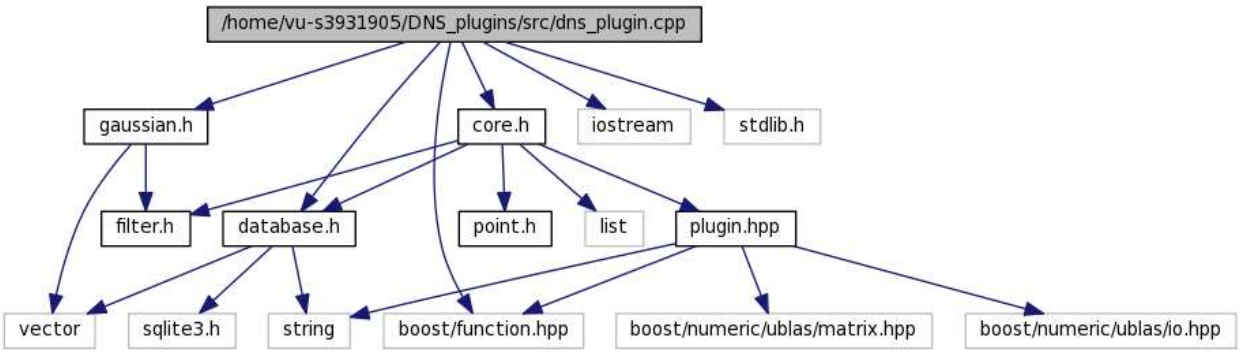
#include <iostream>

Include dependency graph for database.cpp:


/home/vu-s3931905/DNS_plugins/src/database.cpp
database.h
iostream
string
vector
sqlite3.h

7.15 DNS_PLUGIN.CPP FILE REFERENCE

#include "core.h"
#include "database.h"
#include <boost/function.hpp>
#include "gaussian.h"
#include <iostream>
#include <stdlib.h>
Include dependency graph for dns_plugin.cpp:



7.15.1 FUNCTIONS

- int main (int argc, char **argv)

7.15.2 FUNCTION DOCUMENTATION

7.15.2.1 int main (int *argc*, char ** *argv*)

References Database::close(), core::getFilteredVelosity(), core::getFilterWidth(), core::getModelSize(), and FilteredData::p.

```
16 {
17 /*
18 ArgvParser cmd;
19 // init
20   cmd.setIntroductoryDescription("This is foo written by bar.");
21 //define error codes
22   cmd.addErrorCode(0, "Success");
23   cmd.addErrorCode(1, "Error");
24   cmd.setHelpOption("h", "help", "Print this help page");
25
26  //  cmd.defineOption("version", ArgvParser::NoOptionAttribute, "Be verbose");
27   cmd.defineOption("version", "Be verbose", ArgvParser::NoOptionAttribute );
```

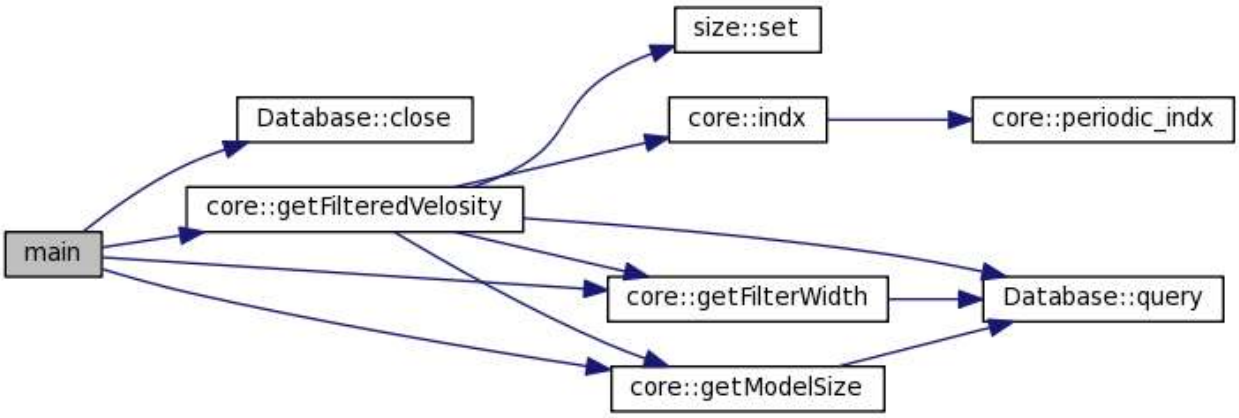
```
28
29
30   cmd.defineOptionAlternative("verbose","v");
31
32  // cmd.defineOption("foo", ArgvParser::OptionRequiresValue, "Fooishness. Default
value: 0");
33   cmd.defineOption("foo",  "Fooishness. Default value:
0",ArgvParser::OptionRequiresValue );
34
35   cmd.defineOption("createDb" );
36   cmd.defineOption("sp", "", ArgvParser::OptionRequired);
37
38
39  // finally parse and handle return codes (display help etc...)
40    int result = cmd.parse(argc, argv);
41
42    if (result != ArgvParser::NoParserError)
43    {
44        cout << cmd.parseErrorDescription(result);
45        exit(1);
46    }
47
48    // now query the parsing results
49    if (cmd.foundOption("foo"))
50    {
51     // string = cmd.optionValue("foo");
52        cout << cmd.optionValue("foo") << endl;
53    }
54    if(cmd.foundOption("createDb"))
55    {
56        cout << cmd.optionValue("createDB") << endl;
57    }
58
59
60 return 0;
61 */
62
63 core *a = new(core);
64
65 string source_dir = string("/home/projects/pVict0004/512/DnsData/");
66 string out_dir = "/home/projects/pVict0004/512/LES/";
67
68 /*
69 out_dir += "LES_2_DB";
70 gaussian g(2);
71 a->createLES_DB(source_dir, g,out_dir );
```

```
72 */
73
74 /*
75 out_dir += "LES_4_DB";
76 gaussian g(4);
77 a->createLES_DB(source_dir, g,out_dir );
78 */
79
80 /*
81 out_dir += "LES_6_DB";
82 gaussian g(6);
83 a->createLES_DB(source_dir, g,out_dir );
84 */
85 /*
86 out_dir += "LES_8_DB";
87 gaussian g(8);
88 a->createLES_DB(source_dir, g,out_dir );
89 */
90
91 /*
92 out_dir += "LES_16_DB";
93 gaussian g(16);
94 a->createLES_DB(source_dir, g,out_dir );
95 return 0;
96 */
97
98 Database *db;
99 db = new Database((char*)"/home/projects/pVict0004/512/LES/LES_2_DB");
100
101 cout << "fw=" << a->getFilterWidth(db) <<endl;
102 cout << "sz=" << a->getModelSize(db) <<endl;
103
104 point p(12,24,32);
105
106 FilteredData  res = a->getFilteredVelosity(p,db);
107
108 cout << "u=" << res.p[0] << endl;
109
110 int y = 10;
111 int z = 10;
112 LIST_DATA ld;
113 LIST_POINTS lp;
114 LIST_DATA::iterator it;
115 for(int x=0; x< 10; x++)
116 {
117 point p(x,y,z);
```

```
118 lp.push_back(p);
119 }
120 a->getFilteredVelosity(lp,ld,db);
121
122 for(it=ld.begin(); it != ld.end() ; ++it)
123 {
124  cout << "u=" << (*it).p[0] << endl;
125 }
126
127
128
129
130
131 /*
132 table res = db->query((char*)"SELECT * FROM data WHERE ind=2;");
133
134 table::iterator it;
135 for(it = res.begin(); it < res.end(); ++it)
136 {
137         row  rw = *it;
138         cout << "Values: (ind=" << rw.at(0) <<
139                         ", u=" << rw.at(1) <<
140                         ", v=" << rw.at(2) <<
141                         ", w=" << rw.at(3) <<
142                         ")" << endl;
143         //cout << "Values: (A=" << rw.at(0) << ", B=" << rw.at(1) << ")" << endl;
144 }
145 */
146 db->close();
147
148 //a->load_plugin();
149 //a->use_plugin();
150 //a->unload_plugin();
151
152 //a->model_stress();
153
154
155 /*
156 Database *db;
157 db = new Database((char*)"/home/vu-s3931905/DNS_plugins/src/Database.sqlite");
158
159 db->query((char*)"CREATE TABLE a (a INTEGER, b INTEGER);");
160 db->query((char*)"INSERT INTO a VALUES(1, 2);");
161 db->query((char*)"INSERT INTO a VALUES(5, 4);");
162 table::iterator it;
163
```

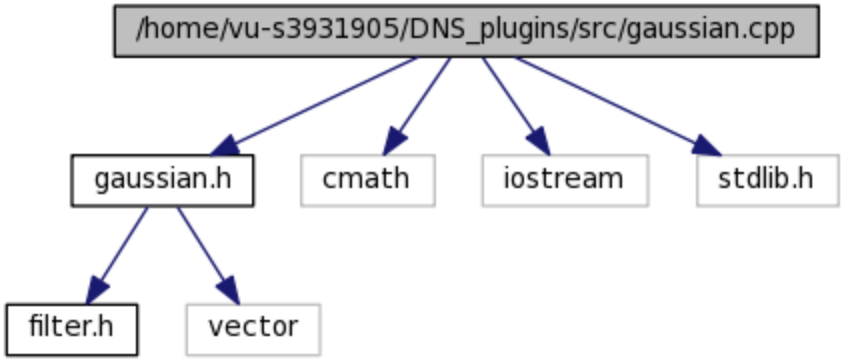
```
164
165
166 table res = db->query((char*)"SELECT a, b FROM a;");
167
168 for(it = res.begin(); it < res.end(); ++it)
169 {
170         row  rw = *it;
171         cout << "Values: (A=" << rw.at(0) << ", B=" << rw.at(1) << ")" << endl;
172 }
173
174 db->close();
175 */
176
177
178
179
180 }
```

Here is the call graph for this function:



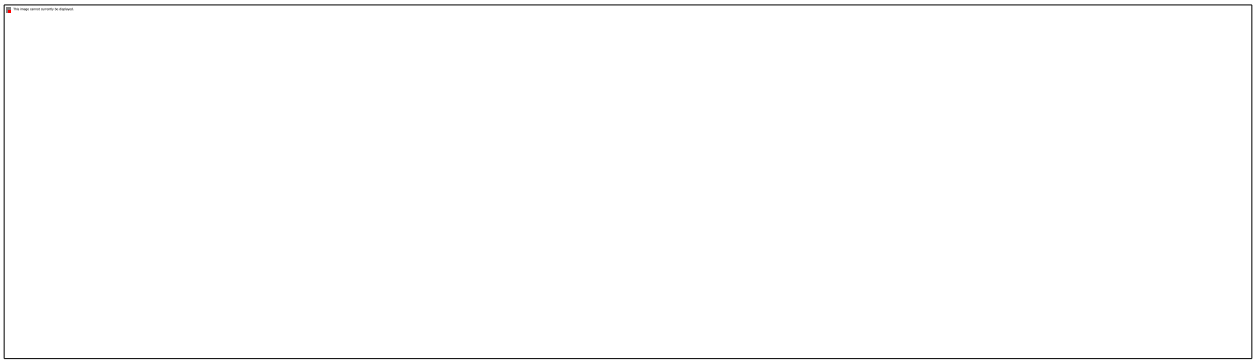
7.16 GAUSSIAN.CPP FILE REFERENCE

#include "gaussian.h"
#include <cmath>
#include <iostream>
#include <stdlib.h>
Include dependency graph for gaussian.cpp:



7.17 PLUGIN.CPP FILE REFERENCE

#include "plugin.hpp"
#include <iostream>
#include "core.h"
Include dependency graph for plugin.cpp:



7.17.1 CLASSES

- class plugins::Smagorinsky

7.17.2 NAMESPACES

- namespace plugins

7.17.3 FUNCTIONS

- plugins::Plugin * construct ()

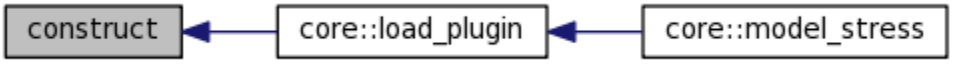
7.17.4 Function Documentation

7.17.4.1 plugins::Plugin* construct ()

Referenced by core::load_plugin().

```
83    {
84        return new plugins::Smagorinsky();
85    }
```

Here is the caller graph for this function:



7.17.5 COMPARATOR OPERATOR

Engineers and fluid dynamicists must have access to a tool that enables the accuracy of alternative LES models. In the existing version of our platform, we verify our results by plotting spectra on log-log coordinates. However, this is a subjective approach, and we are implementing a quantitative method that will result in a simple index that will inform users of the accuracy of their models. The idea is to quantify the similarity of patterns based on principal component analysis, and particularly by making use of the Pearson correlation coefficient [11].

The squared PSX correlation coefficient r^2 can be calculated by dividing covariance of ss_{ab} of two spectra *a* and *b* by the product of their standard deviations ss_{aa} and ss_{bb}

$$\boldsymbol{r^2 = \frac{SS_{ab}^2}{SS_{aa}SS_{bb}}} \tag{7.4}$$

where

$$ss_{aa} = \sum_{i=1}^{n} (a_i - \overline{a})^2 \tag{7.5}$$

$$ss_{bb} = \sum_{i=1}^{n} (b_i - \overline{b})^2 \tag{7.6}$$

$$ss_{ab} = \sum_{i=1}^{n} (a_i - \overline{a})(b_i - \overline{b}) \tag{7.7}$$

The comparison process proceeds by calculating the distribution of a macroscopic property such as shear stress, the rate of energy dissipation and so on. This gives rise to sz×sz×sz values where sz is the dimension of our domain. The next step which will be invisible to the user is to create the FFT of the values that reduce the number of components to sz numbers. The final step is to calculate the Pearson correlation coefficient that provides a single measure of the quality of the LES model.

7.18 RESULTS AND PRACTICAL USAGE EXAMPLES

To demonstrate the operability of our approach we compared the spectra generated by DNS and those arising from LES with a range of filter widths. The example chosen makes use of data

accessible from the Johns Hopkins University database of DNS solutions [12], and we consider one that concerns forced isotropic turbulence. The problem is described in [12] thus:

Direct numerical simulation using $1{,}024^3$ nodes.

The Navier-Stokes equations are solved using a pseudo-spectral method.

Energy is injected by keeping constant the total energy in shells such that |k| is less or equal to 2.

After the simulation has reached a statistically stationary state, 1,024 frames of data with three velocity components and pressure are stored in the database. Extra time frames at the beginning and at the end have been added to be used for temporal-interpolations.

The Taylor-scale Reynolds number fluctuates around $Re_\lambda \sim 433$

There is one dataset with 1024 time-steps available, for time *t* between 0 and 2.048 (the frames are stored at every 10 time-steps of the DNS). Intermediate times can be queried using temporal interpolation.

In our problem, this requested the loading of a 512×512×512 domain and for all filter widths in the range 2 to 32 we calculated filtered spectra. Figure 7.5 shows that the stress components calculated by DNS and arising from a LES solution with a filter width of four are in close agreement; hence a LES solution would be expected to provide accurate solutions in about two orders less time than a DNS solution.

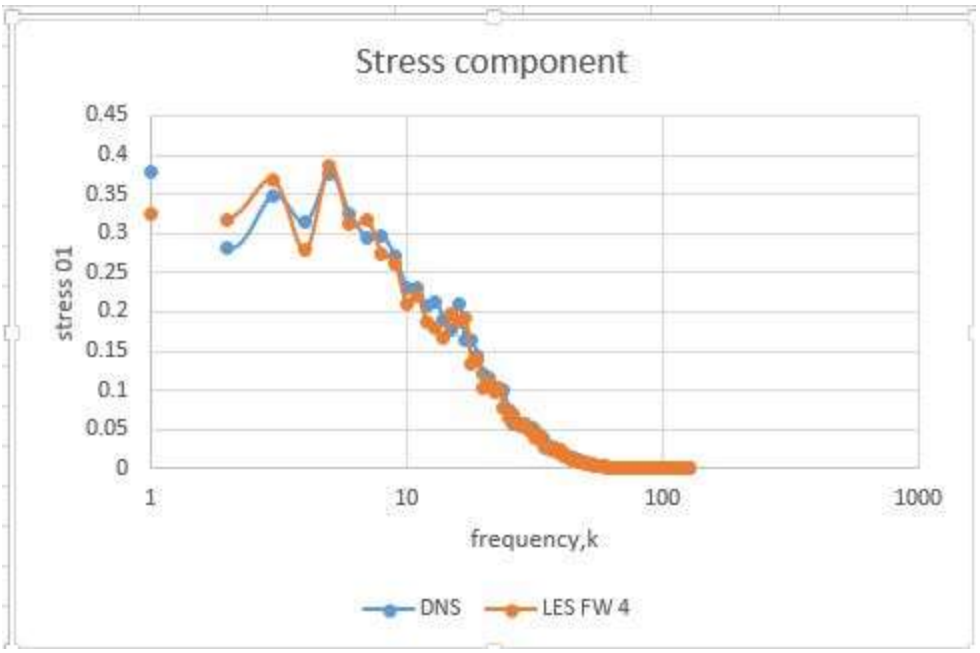


Figure 7.5 A comparison of stress components obtained from DNS and those obtained with a Smagorinski filter of width 4

Figure 7.6 demonstrates the transfer function of the velocity as a function of the filter width. If the LES and DNS solutions were coincident, the transfer function would be unity; however, it can be seen that this ideal is approached as the filter width is reduced.

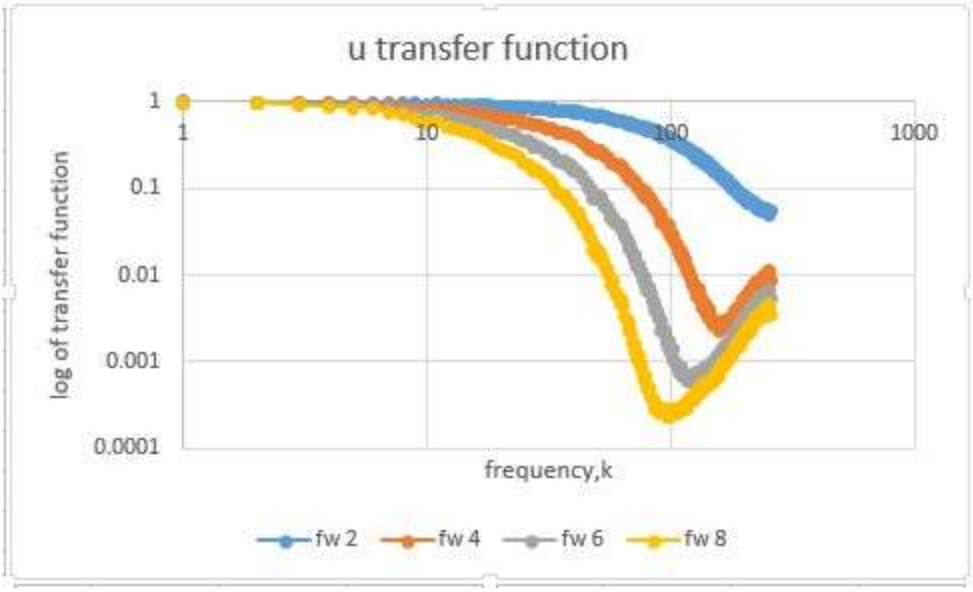


Figure 7.6 The transfer function calculated with filter widths of 2, 4, 6 and 8.

7.19 CONCLUSIONS

Databases of solutions of the Navier-Stokes equations generated by direct numerical simulation are necessarily broad. They are nevertheless handy for developing new, more practical yet accurate approximations of the Navier-Stokes equations. Motivation of this work is to develop an intellectual framework whereby CFD practitioners have an easy-to-use tool that enables them to evaluate their LES models. At the heart of the method database technology that harnesses the following ideas::

- Very large volumes of data are manipulated by storing variables in heap memory.
- A readily available client-based database engine implemented
- Fast-Fourier transforms algorithms are used to ensure that analyses are carried out efficiently.

The principal idea behind the method is that a core is defined that contains the ‘know-how’ associated with accessing and manipulating data, and which operates independently of a plugin. This enables users to propose LES models and obtain results almost instantaneously**.** As a result, users can operate on large sets of data and obtain results almost instantaneously.

8 CONCLUSIONS

Modern engineering requires practical solutions of turbulent flows for models with complex geometries and boundary conditions. This entails dealing with large amounts of data and time-consuming calculation. In this work, we demonstrate how both of these requirements can be satisfied.

In the first part of the thesis, we develop paradigm and methodology for speeding up massive calculations by parallelisation of Navier Stokes equations. We discuss and demonstrate new phenomena which only appear in the parallel world and explain how they can be used in turbulent flow simulation.

Parallelisation can be defined as a process of discovery of independent parts of the task. There are several tasks which are naturally parallel. For example, this is the case in the area of an image. An image is essentially an array of independent pixels with corresponding properties, and it can be argued that the parallelisation of the image is a trivial task. However, there are many tasks such as rendering in computer visualisation (Eilemann, 2019), brute force searches Loesch (1990) and so on. Moler (1986) coined the phrase "embarrassingly parallel" to describe problems that are parallel and easy to solve.

In contrast, performing parallelisation of applications in the domain of computational fluid dynamics is relatively quite challenging (Simon, 1991), Griebel & Zaspel, 2010 and Hauser & Williams, 1992). This is because CFD simulations require solving several interdependent equations with many parameters that are also interdependent. It is significant to note that a Google Scholar search for 'Navier Stokes Parallelization' limited to the years 2018-2019 returns more than 4000 articles. This is just another prove that Navier Stokes Parallelization is not as trivial and so so many researchers still working on the problem

The term 'paradigm' is defined in the Cambridge English Dictionary as – "a model of something, or an obvious and typical example of something":

What we have achieved in this thesis is to devise a paradigm such that several necessary steps are carried out in parallel, but the specific implementation of

each step is left open. Finally, we have demonstrated this paradigm in a specific sequential CFD application and demonstrated how this application was converted to parallel.

Another point is the development of parallel applications leaves many ordinary developers in the dark. They are entirely unfamiliar with the subtleties of parallelisation. This was my motivation for writing Chapter 3, in which I discuss in detail several scenarios and terminologies that appear only in the parallel world of software development. These include parallel overheads, dead-locks, mutexes, threads and thread functions, thread safety, locks, synchronisations, and so on. It is advanced software concepts such as these that are implemented in the thesis. All of these phenomena exist only in the parallel world of software development.

We developed a standalone thread pool class which we inject into a sequential channel flow solver to execute those regions of the code that are computationally intensive. We have demonstrated that increasing the number of threads to two is speeding up of calculations more than double. A significant contribution of this work is that we exploit the benefits of encapsulation which allow multiple users to work in the same space.

Multithreading applications do not always run faster. This is because of overheads in creating threads and communication between the threads and attempts to run the sequential algorithm in parallel which requires using many mutexes. Difficulties in decomposing problems into a parallel form often require considerable intellectual effort to overcome. There are many pitfalls in the parallel world, and they have been discussed in great detail. The fact that our paradigm enables us to reduce calculation time for channel flow is evidence that the classes we have developed can be reused for other serial Large Eddy Simulation applications**.**

We developed an application that simulates channel flow by performing calculations in a parallel, and I compared the results with an application that simulates channel flow by implementing a serial paradigm.

Code validation is not to determine differences. The aim is to show there are no differences in the results (Foster, I., Olson. R., & Tuecke, S., 1992). This was carefully done by comparing the outputs of each function and member classes. Once that is successfully achieved, there is no need to run the same code repeatedly for the different sizes of systems because system size does not influence calculation logic.

Also, encapsulation can be described as the protection of data elements and the provision of methods with which to access the data, thus "encapsulating" the state of an object with the actions that can be performed upon it. However, the benefits of this concept are much more comprehensive. It not only allows one to carry out calculations in parallel, but it also allows one to perform development in parallel (Cantor, 1998), Culler, Singh, J. & Gupta, 1999)

For example, a research team may have a mammoth task to develop new classes for the IT industry. Each member of the team works on developing one common object, but each member has access to the relevant private areas of the task. The concept of encapsulation assures there will be no clashes. Even application testers can work in parallel and write their scripts without waiting until the developers complete their coding. Ultimately, new objects can be created, tested and validated by a small team, working in parallel, and the approach enables tasks to be completed in shorter times. This is an underlying approach adopted in this thesis.

In this work, the method of local parallelisation has been applied to channel flow. However, the technique is quite general and powerful, and it can be used to a myriad of practical and research problems that involve turbulent flows.

The direct numerical solutions of the Navier-Stokes equations are obtained using short time and length scales. As a result, these solutions inevitably contain prodigious quantities of data. In this work, we have developed an approach to rapidly and conveniently analyse the solutions. We demonstrate a developmental approach to deal with large sets of data. We have created a highly efficient platform that is intended to be easy to use by the scientific community to devise and test their sub-grid LES models against the results of DNS, Johns Hopkins database of DNS solutions was used as for comparison. To help scientists and engineers to evaluate their LES models, we present a comprehensive comparator operator to quantify the accuracy of the models. Furthermore, the method releases the researcher from the need to write a comprehensive code because the LES models can be implemented as plugins.

This work has presented an intellectual framework whereby CFD practitioners can readily and quickly examine the accuracy of new models they might wish to propose. The method is based on database technology and includes the following concepts:

- Use and manipulation of heap memory to handle vast volumes of data;
- The implementation of a client-based database engine; and
- The incorporation of efficient fast-Fourier transforms algorithms.

The package is implemented on an HPC cluster. An idea permeating the methodology is that a core is defined that contains the 'know-how' associated with accessing and manipulating data, and which operates independently of a plugin. This enables users to propose LES models and obtain results almost instantaneously.

Heap memory manipulation presented in Chapter 6.2 shows there is core class and there is a class member allocate_3Darray, this member returns a pointer to a chunk of heap memory. Passing this pointer as a parameter to a function allows the use as a statically defined three-dimensional array. This approach overcame stack data type memory limitations and manipulates extensive data arrays and work with them like a usual indexed array. As to FFTW, Running FFT in parallel is another trigger that we use to speed up simulation processes. FFTW (Frigo et al., 1998) is an open-source implementation of FFT. At the moment, it is still considered the fastest implemented FFT algorithm. FFTW has inbuilt multithreaded capabilities which make encapsulating it in DNS code relatively easy. Using parallel stand-alone libraries will probably be an increasingly popular way to speed up sequential processes.
This concept was accepted and presented by me on First Thermal and Fluids Engineering Summer Conference, 9-12 August 2015, New York, NY, USA
https://www.astfe.org/conferences/tfesc/TFESC_Conference_Technical_Program.pdf

This was presented to JHTDB researchers and received with great interest, as it significantly speeds up using JHTB and delivers practical results to scientists almost immediately.

9 INDEX

10 TABLE OF FIGURES

11 NOMENCLATURE

C_s Smagorinsky constant

f Arising from body force, m/s^2

G Filter function

p Pressure, Pa

S Rate of strain tensor, s^{-1}

t Time,s

u Velocity, m/s

x Distance in x-direction, m

y Distance in the y-direction, m

z Distance in the z-direction, m

N Number of computational nodes

L Dimension of the computational domain

ε Rate of kinetic energy dissipation, J/kg

Re Reynolds number

Re_λ Taylor-scale Reynolds number

Greek symbols

Δ Width of filter, Laplacian, Change in the value of the variable, m

$\delta_{i,j}$ Dirac delta function

μ Dynamic viscosity, kg/(ms)

v Kinematic viscosity, m^2/s

ρ	Density, kg/m^3
φ	General filter function

Subscripts

i,j	Denotes *i* and *j* coordinates

Acronyms

sgs	Sub-grid scale
LES	Large eddy simulation
RANS	Reynolds averaged Navier-Stokes equations
DNS	Direct numerical simulation
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
HPC	High-Performance Computer
OOP	Object-Oriented Programming
OOA	Object–Oriented Analysis

12 REFERENCES

Arbenz, P, & Obrist, D, (2018) *Comparison of Parallel Time-Periodic Navier-Stokes Solvers*, Parallel Processing and Applied Mathematics,2018

Barney, B. Introduction to Parallel Computing. Retrieved from https://computing.llnl.gov/tutorials/parallel_comp/#Terminology,

Blaise. B, (2010) *POSIX Threads Programming*, Lawrence Livermore National Laboratory, 2010, https://computing.llnl.gov/tutorials/pthreads/

Bodis, L. (2007). *Quantification of Spectral Similarity.* (Computer Science), Babes-Bolyai University, Romania. (17361)

Booch, G. (1982). Object-oriented design. *Ada Lett., I*(3), 64-76. doi:10.1145/989791.989795

Canuto, C. (1988). *Spectral methods in fluid dynamics*: Springer-Verlag.

Cantor, M., (1998). Object-Oriented Project Management with UML, ISBN: 0471253030 ,1998, Publisher: John Wiley & Sons, Inc.

Chapman, D. (1979). Computational Aerodynamics Development and Outlook. *AIAA Journal, 17*(12), 1293-1313. doi:10.2514/3.61311

Culler, D. & Singh, J. & Gupta, A, (1999) . *Parallel Computer Architecture* - A Hardware/Software Approach. Morgan Kaufmann Publishers, 1999. ISBN 1-55860-343-3, pg 15

Dahl, O.-J. (2004). *The Birth of Object Orientation: the Simula Languages.*

Owe, S. Krogdahl, & T. Lyche (Eds.), *From Object-Orientation to Formal Methods:* Essays in Memory of Ole-Johan Dahl (pp. 15-25). Berlin, Heidelberg: Springer Berlin Heidelberg.

Darlington, J., Ghanem, M., Guo, Y., & To, W. T. (1996).), Guided Resource Organisation in Heterogeneous Parallel Computing. *Journal of High-Performance Computing, 4* (1), 13-23.

Drysdale, D. (2007). *High-Quality Software Engineering: Lessons from the Six-Nines Worlds*: Lulu.com.

Eilemann, S. (2019), *Parallel Rendering and Large Data Visualization,* PhD thesis, University of Zurich, Neuchatel, NE, Switzerland, https://arxiv.org/pdf/1902.08755.pdf

Elke, J. (2004). Origin of the Virtual Memory Concept. *IEEE Annals of the History of Computing, 26*(4), 71-72.

Feynman, R., Leighton, R., & Sands, M. (1963) The Feynman lectures on physics (2nd ed.). Oxford: Addison-Wesley World Student Series.

Foa, E., Cashman, L., Jaycox, L., & Perry, K. . (1997). The validation of a self-report measure of PTSD: The Posttraumatic Diagnostic Scale. *Psychological Assessment, 9,* 445-451.

Foster, I., Olson. R., & Tuecke, S., (1992) Productive Parallel Programming: The PCN Approach. Scientific Programming Volume 1, Issue 1, 51-66, 1992

Foundation, N.. (2014). Johns Hopkins Turbulence Databases – *Forced Isotropic Turbulence.* . Retrieved from http://turbulence.pha.jhu.edu/datasets.aspx

Fox, L., & Parker, I. (1968). *Chebyshev polynomials in numerical analysis*. Oxford: University Press.

Frigo, M., & Johnson, S. G. (1998, 12-15 May 1998). *FFTW: an adaptive software architecture for the FFT.* Paper presented at the Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on.

Frigo, M., & Johnson, S. G. (2005). The Design and Implementation of FFTW3. *Proceedings of the IEEE, 93*(2), 216-231. doi:10.1109/JPROC.2004.840301

Frumkin, M. & Schultz, M. & Jin, H. & Yan,J. (2003) *Performance and scalability of the NAS parallel benchmarks in Java, Proceedings International Parallel and Distributed Processing Symposium,* Nice, France, 2003, pp. 6 -8.doi: 10.1109/IPDPS.2003.1213267

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., . . . Rossi, F. (2009). *GNU Scientific Library Reference Manual* Retrieved from UK:

Gibson, J. (2014). *Channelflow: a spectral Navier–Stokes simulator in C++*.Retrieved from

www.channelflow.org.

Gnanaskandan, A., & Mahesh, K. (2016). Large Eddy *Simulation of the transition from sheet to cloud cavitation over a wedge*. International Journal of Multiphase Flow, 83, 86-102.

Gottlieb, D., & Orzag, S. (1977). *Numerical Analysis of Spectral Methods: Theory and Application,* : Philadelphia, PA.

Griebel, M. & Zaspel. P., (2010) A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier Stokes equations. *Computer Science-Research and Development, 2010 - Springer*

Gropp,W. & Kaushik, D. & Smith, B., (2001), *High-performance parallel implicit CFD,* parallel Computing 27(4) 337-362

Grossman, I. (2015) Begell house, *A PLATFORM THAT ACCEPTS SUB-GRID MODELS AS PLUGINS TO ENABLE THE TESTING OF LES MODELS AGAINST DNS DATA*, http://search.begellhouse.com/index.php

Grossman, I. (2015) First Thermal and Fluids Engineering Summer Conference, 9-12 August 2015, New York, NY, USA https://www.astfe.org/conferences/tfesc/TFESC_Conference_Technical_Program.pdf

Haldar, S. (2015). *SQLite Database System Design and Implementations*: https://books.google.com/.

Hauser, J., & Williams, R. (1992). Strategies for parallelizing a Navier Stokes code on the Intel touchstone machines. *Journal for numerical methods in fluids, 1992*

Hollman, D. & Bennett, J. & Kolla, H. & Lifflander, J. & Slattengren, N. & Wilke, J. (*2016)Metaprogramming-Enabled Parallel Execution of Apparently Sequential C++ Code, 2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2),* Salt Lake City, UT, 2016, pp. 24-31. doi: 10.1109/ESPM2.2016.009

Imlay, S. (2017, December 14). Tecplot. [Blog] *Why One Trillion Cells?.* Available at: http://www.tecplot.com/blog/2014/12/03/one-trillion-cells/ [Accessed 2 Sep. 2016]

Jacobson, I. (1992). *Object-oriented software engineering*: ACM.

John, V. Comput Visual Sci (1999) 1: 193.https://doi.org/10.1007/s007910050 *Journal of Fluid Mechanics, 774*, 395-415.

Katz, A. & Jameson, A.(2010), *Meshless Scheme Based on Alignment Constraints*, AIAA Journal,Vol.48. No. 11, November 2010

Kay, A. (1972). *A Personal Computer for Children of All Ages*. Paper presented at the Proceedings of the ACM annual conference - Volume 1, Boston, Massachusetts, USA.

Kay, A. C. (1993). *The early history of Smalltalk*. Paper presented at the The second ACM SIGPLAN conference on History of programming languages, Cambridge, Massachusetts, USA.

Kerr, R.., & Kimura. Y., (editors) 2000: *Developments in Geophysical Turbulence. Proceedings of the IUTAM* symposium at the National Center for Atmospheric Research, Boulder, CO 16-19 June 1998. Kluwer Academic Publishers, Dordrecht

Knuth, D. (2007). *Fundamental Algorithms, Section 2.5: Dynamic Storage Allocation, .*

Krist, S., & Zang, T. (1987). *Numerical simulation of channel flow transition* (NASA-TP-2667, L-16204, NAS 1.60:2667). Retrieved from Hampton VA United States:

Kroll, N., & Rossow, C. (2015). Current status and challenges in CFD at the DLR Institute of Aerodynamics and Flow Technology. Retrieved from. https://www.grc.nasa.gov/hiocfd/wp-content/uploads/sites/22/AIAA2015_Challenges_for_CFD_DLR_Kroll.pdf

Krüger, J., & Westermann, R. (2003). Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003, 22*(3), 908=916.

Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers, C-28*(9), 690-691. doi:10.1109/TC.1979.1675439

Lanczos, C. (1938). Trigonometric Interpolation of Empirical and Analytical Functions.

Journal of Mathematics and Physics, 17(1-4), 123-199. doi:10.1002/sapm1938171123

Lavington, S. (1998). *History of Manchester Computers*. Swindon: British Computer Society.

Lavington, S. H. (1975). *A history of Manchester computers*: NCC Publications.

Lee, M., & Moser, R. . (2015). Direct numerical simulation of turbulent flow up to Re≈5200

Li, Y., Perlman, E., Wan, M., Yang, Y., Meneveau, C., Burns, R.,Eyink, T.(2008). A public turbulence database cluster and applications to study the Lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence, 9*, 1-29.

Loan, C. (1992). *Computational Frameworks for the Fast Fourier Transform.* Philadelphia: SIAM Press

Loesch, H & Remscheid, A (1990) *Brute force in molecular reaction dynamics: A novel technique for measuring steric effects*,J. Chem. Phys. **93**, 4779 (1990); https://doi.org/10.1063/1.458668

Losavio, F., Matteo, A., & Schlienger, F. (1994). Object-oriented methodologies of Coad and Yourdon and Booch: comparison of graphical notations. *Information and Software Technology, 36*(8), 503-514. doi:https://doi.org/10.1016/0950-5849(94)90028-0

Lyle B. (1966). Algorithm 277. *Computation of Chebyshev series coefficients, 9 (2), 86–87.* doi: doi:10.1145/365170.365195

McCalpin, J. & Moore, C. & Hester, P. (2011). *The Role of Multicore Processors in the Evolution of General-Purpose Computing*. CTWatch Quarterly. 3. 18-30.

Menabrea, L.(1842). *Sketch of the Analytical Engine Invented by Charles Babbage, Esq*: Richard and John E. Taylor.

Mitchell, D., Arun N. (1988). *Reconstruction filters in computer-graphics* Paper presented at the International Conference on Computer Graphics and Interactive Techniques.

Moler, C. (1986). Heath, Michael T. (ed.). *Matrix Computation on Distributed Memory Multiprocessors*. *Hypercube Multiprocessors*. Society for Industrial and Applied Mathematics, Philadelphia. 1986, ISBN 978-0898712094.

Moore, G. (1965). Cramming more components onto integratedcircuits *Electronics, 86*(1),

114-117.

Muriel, A. (2010) *An Exact Solution of the 3-D Navier-Stokes Equation*, Department of Electrical Engineering Columbia University and Department of Philosophy Harvard University, Retrieved from

Muriel, A. and Dresden, M., *An Exact Solution of the 3-D Navier-Stokes Equation* Physica D 101, 297 (1997).

Naveed,A, & Ulrich,W, (2017) *An assessment of some solvers for saddle point problems emerging from the incompressible Navies-Stokes equations*, Computer Methods in Applied Mechanics and Engineering, 2017

Norberg. A, & O'Neill. J (1996) *Transforming Computer Technologies, Johns Hopkins Press,Baltimore and London*, MAIN QA 76 .17 N67 1996

Nvidia, (2007), *CUDA*, https://www.nvidia.com/object/cuda_home_temp.html, Parallel CFD conference, (2018), http://www.indiana.edu/~parcfd18/

Perlman, E., Burns, R., Li, Y., & Meneveau, C. (2007). *Data exploration of turbulence simulations using a database cluster*. Paper presented at the Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada.

Pope, S. (2004). Ten questions concerning the large-eddy simulation of turbulent flows *New Journal of Physics, 6*(35), 8-9.

Premerlani, W. J., Rumbaugh, J. E., Blaha, M. R., & Varwig, T. A. (1990). An object-oriented relational database. *Commun. ACM, 33*(11), 99-109. doi:10.1145/92755.92772

Reveillon, J., Pera, C., & Bouali,Z., Examples of the potential of DNS for the understanding of reactive multiphase flows.*International journal of spray and combustion dynamics · Volume. 3 Number . 1 .2011 – pages 63–92*

Reynolds, O. (1895). On the Dynamical Theory of Incompressible Viscous Fluids and the Determination of the Criterion. *Philosophical Transactions of the Royal Society of London. (A.), 186*, 123-164. doi:10.1098/rsta.1895.0004

Richardson, L. The problem of contiguity: An appendix to Statistic of Deadly Quarrels. General systems: Yearbook of the Society for the Advancement of General Systems Theory. 1961

Sanders, M. J., Leslie, M., & Catlow, C. R. A. (1984). Interatomic potentials for SiO2. *Journal of the Chemical Society, Chemical Communications*(19), 1271-1273. doi:10.1039/C39840001271

Sarwar, M., Cleary, M.J., Moinuddin, K.A.M, Thorpe G. R. (2017). International Journal of Heat and Fluid Flow. *On linking the filter width to the boundary layer thickness in explicitly filtered large eddy simulations of wall bounded flows*, 73-89

Scott .M, (2009) Semantic Analysis, Programming Language Pragmatics. 2009. pp. 175-211

Shelly, A. (2008). Multicore Programming (Multiprocessing) Visual C++ Tips: Design Guidelines.

Shelly, A. (2010). HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions. Retrieved from https://www.revolvy.com/main/index.php?s=Object%20Oriented&item_type=topic

Shen, J. P., & Lipasti, M. H. (2013). *Modern processor design : fundamentals of superscalar processors*. Long Grove: Waveland Press.

Silberschatz A., Gagne G. G., & Galvin P. (2012). *Operating System Concepts* (Ninth ed.). United States: John Wiley & Sons, Inc.

Simon, H. (1991) *Partitioning of unstructured problems for parallel processing,* Computing Systems in Engineering Volume 2, Issues 2–3, 1991, Pages 135-148

Singh, P, & Singh, A. (2018). *Growth Trend in Global Big Data Research Publications as Seen From SCOPUS Database*. Professional Journal of Library and Information Technology, 8 (1),49-61. **(ISSN: 0976-7574),** https://www.academia.edu/38667284/Growth_Trend_in_Global_Big_Data_Research_Publications_as_Seen_from_SCOPUS_Database

Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., & Mavriplis, D. (2014). *CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences*. Retrieved from United States:

Smagorinsky, J. (1963). General circulation experiments with the primitive equations. *Monthly Weather Review, 91*(3), 99-164.

Smits, A., & Marusic, I. (2013). Wall-bounded turbulence. *Physics Today, 66*(9), 25-30.

Stephen, B. P. (2004). Ten questions concerning the large-eddy simulation of turbulent

flows. *New Journal of Physics*, *6*(1), 35.

Strazdins, P. E. (2012, 21-25 May 2012). *Experiences in Teaching a Specialty Multicore Computing Course.* Paper presented at the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

Sutter. H, (2005) The free lunch is over. *A Fundamental Turn Towards Concurrency in Software*, Dr. Dobb's Journal,38(3),March 2005

Tennekes, H., & Lumley, J. L. (1972). *A First Course in Turbulence*: MIT Press.

Trebotich, D. & , Straalen, B. & D Graves, D. & Colella,P. (2008), *Performance of embedded boundary methods for CFD with complex geometry,* © 2008 IOP Publishing Ltd
Journal of Physics: Conference Series, Volume 125, Number 1

Tritton, D. J. (1988). *Physical Fluid Dynamics* (Second Edition ed.). Oxford: Oxford University Press.

Van Heesch, D. Doxygen: Generate documentation from source code. Retrieved from http://www.stack.nl/~dimitri/doxygen/

Van Loan, C. (1992). *Computational Frameworks for the Fast Fourier Transform*: Society for Industrial and Applied Mathematics.

Vyšohlíd, M. (2007). *Large Eddy Simulation of crashback in marine propellers.* (Aerospace Engineering), University of Minnesota, United States -- Minnesota. (3266766)

Walshaw, C. & Gross, M. & Everett, M. (1997), *Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes*,Journal of Parallel and Distributed Computing,Volume 47, Issue 2, 15 December 1997,pp. 102-108

Wang,Y. & Baboulin,M. & Rupp,K. & Le Maitre, O. (2014) *Solving 3D incompressible Navier-Stokes equations on hybrid CPU/GPU systems*; Talk: High Performance Computing Symposium (HPC), Tampa, Florida, USA; 2014-04-13 - 2014-04-16; in "*HPC '14 Proceedings of the High Performance Computing Symposium*", (2014), 1 - 8.

Westermann, K. (2005). *Linear algebra operators for GPU implementation of numerical algorithms*. Paper presented at the International Conf. on Computer Graphics and Interactive Techniques.

Wilson, P. R., Johnstone, M. S., Neely, M., & Boles, D. (1995). Dynamic storage allocation: A survey and critical review. In H. G. Baler (Ed.), *Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings* (pp. 1-116). Berlin, Heidelberg: Springer Berlin Heidelberg.

Yang, H., Zimmerman, A., & Lehner, L. (2015). Turbulent Black Holes. *Physical Review Letters, 114* (8), 081101.

You, D., & Moin, P. (2007). A dynamic global-coefficient subgrid-scale eddy-viscosity model for large-eddy simulation in complex geometries. *Physics of Fluids, 19*(6), 065110.

Yuan, J., & Piomelli, U. (2015). Numerical simulation of a spatially developing accelerating boundary layer over roughness. *Journal of Fluid Mechanics, 780*, 192-214.