Hybrid Parallelization of a Spatial Direct Numerical Simulation

Pedro Espinheira Alves
pedro_e_alves@hotmail.com

Instituto Superior Técnico, Universidade de Lisboa, Portugal

November 2016

Abstract

High Performance Computing is important in the context of Computational Fluid Dynamics. The performance of Direct Numerical Simulations is a limiting factor in the research of turbulence. To improve the performance of simulations, parallelization is a necessity. Hybrid parallelization has the potential to improve performance of more conventional parallelization of simulation programs. In this work, a hybrid parallelization with MPI+OpenMP is implemented in a program of direct numerical simulation of a spatial turbulent planar jet. Performance is analyzed comparatively with hybrid cases and pure MPI cases, and profiling is done. One 75 million point simulation is completed to demonstrate the feasibility of the hybrid parallelization of this program. The hybrid program shows a superior performance compared with the pure MPI implementation in the tested cases.

Keywords: Hybrid Parallel Programming, High Performance Computing, Direct Numerical Simulation, Planar Jet, Turbulence, MPI+OpenMP Parallel Programming

Introduction

One of the ways to study how a fluid behaves is by simulating it in a computer. By simulating a fluid it becomes possible to obtain high level detail about a certain fluid flow like complete velocity and pressure fields. These flowfields would otherwise be impossible to measure with a physical setup.

The computational power needed to perform a Direct Numerical Simulation (DNS) is extreme in most real cases, and unattainable with current technology for flows of engineering interest. Usually leaving the study of turbulence through DNS reserved to specific cases of scientific interest.

For the handling of these demands of computer power, High Performance Computing becomes a requisite. Since the last decade the processor frequency has stagnated at a certain value, with the processing power offered increasing by the use of numerous processing units in parallel.

This entails a change in approach, which demands an increase in the efficiency of parallelization.

Background

Governing Equations

To analyze a fluid flow, two fundamental physics laws are of fundamental importance. Those are the conservation of mass and the conservation of momentum.

For an incompressible fluid, which is the case of flows with low Mach numbers, the mass conservation law can be expressed by equation 1.

$$\frac{\partial u_i}{\partial x_i} = 0$$  \hspace{1cm} (1)

And the momentum conservation of a Newtonian fluid by the Navier-Stokes (equation 2).

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = - \frac{1}{\rho} \frac{\partial P}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j^2}$$  \hspace{1cm} (2)

with the kinematic viscosity being $\nu = \frac{\mu}{\rho}$.

Turbulence

There is no easy definition of turbulence. From a mathematical point of view, turbulence is the chaotic behavior of flows originated in the nonlinear term of the Navier Stokes partial differential equations. And the result is seemingly random, chaotic fluctuations in the velocity, pressure and other flowfields.

The main parameter revealing the appearance of turbulence is the Reynolds number(Re), defined by equation 3.

$$Re = \frac{U_{\text{ref}} \rho}{\mu} = \frac{U_{\text{ref}}}{\nu}$$  \hspace{1cm} (3)

Turbulence is present in almost every fluid flow, both natural and man made.

DNS

The description of flow characteristics, is a complicated task, and the solving of the full Navier-Stokes equations is a...
for flows of engineering interest with high Re numbers, walls and complicated geometries is unattainable with current technology due to the extreme computational needs [17].

However, for scientific interest cases, the solving of the fluid flowfields to the finest detail without turbulence models is possible in some cases. This method is called Direct Numerical Simulation (DNS), because as the name implies, there is a direct simulation with no turbulence models that can affect the fluid physics.

The limits imposed by the computational resources will keep DNS from being applicable to engineering flows for some years, as the range of scales needed to simulate without turbulence models depend on the number of grids points with an order of $O(Re^{3/4})$, but DNS simulations have been increasing in both resolution and Reynolds number at great pace, with some asserting that in some decades a point where DNS will allow for simulations of fluids not occurring on nature [3].

Turbulent Plane Jet
Turbulent jets are part of a large family of free shear flows. Free shear flows, are flows developing far away from the influence of wall boundaries, characterized for developing along a preferred direction.

In the case of DNS applications, the main objective is usually the study of turbulence, and a plane jet offers a possibility to study turbulent flows of scientific and engineering interest.

High Performance Computing, Parallel Computing, and CFD
Microprocessor frequency increased at a steady rate from the 60s to around 2004, but the laws of physics took their toll on this evolution. The CPU frequency stagnated at around 3GHz. As further miniaturization of the components will not yield faster processing because of the impossibility of dissipating the heat generated from electrical resistance faster [15].

The solution to bigger computational power is to use various processing units working on the same task. This gives rise to the necessity of parallel programming.

Nowadays most high performance machines are clusters of shared memory nodes [19].

Since June of 2016, the record holder is the Sunway Taihulight, with a performance of 93 Petaflop/s on the selected test used, Linpack, and a peak 125.4 Petaflop/s. The Sunway Taihulight is composed of 40 960 nodes, each one with a 260-core chip with a performance of 3 teraflop/s. This adds up to a total of 10 649 600 cores and the announced peak 125 Petaflop/s [6].

Distributed memory vs. shared memory parallelization
The most obvious way to imagine a parallel execution code is to visualize a shared memory that is accessible by various processors. This implies that all threads or processes see and have access to the same memory. It should be acknowledged that in most HPC cases this is an idealization.

OpenMP can be considered the standard Application Programming Interface (API) for shared memory machines, relying on threads [4]. OpenMP uses the Fork-Join model, where the master thread is responsible by creating other threads and opening a parallel part in this way, where the work is distributed between threads, usually in parallel loops or sections, synchronizing at the end of this parallel area, and joining the threads again so the code proceeds in serial mode again.

The shared memory and threads model has some limitations, the most direct being that all processes or processors demand access to the same memory.

In practice this means that High Performance Computing applications of shared memory models are constrained to single nodes.

To program distributed memory machines to harvest their potential a distributed memory approach is needed. Since multiple memories are used, the handling of communications is a fundamental part of this programming.

Message Passing Interface, or MPI, is the standard API used to parallelize codes that run on distributed memory architectures. With MPI the communications have to be explicitly handled by the user. A pure MPI implementation binds one MPI process per physical core. MPI is a flat model, considering each separate memory equally distant and not accounting for differences between intra and internode communications. This is an important limitation that implies unnecessary communications between processes.

Other limiting factors with the MPI approach are the rigidity with respect to load balancing and the increase of memory requirements with the increase in processes number.

Hybrid parallelization - the next step?
Both shared memory and distributed memory models have advantages and disadvantages. The solution to this problem is to use codes that are hybrids of distributed memory parallelism with shared memory parallelism. Developing hybrid programs that use MPI to communicate between distributed memories and OpenMP to share memories among various processors is a logical step when using multicore clusters [5].

At present time, nodes typically have more than one core, meaning that multiple threads running on each node are the best way to harvest the processing
power available. Starting one MPI process per core is a waste of memory and managing high numbers of MPI processes seen in the biggest clusters is not an easy task [2].

The distributed memory parallelism is used to communicate between physically separated memories, in the communication part of the code. Whether this should be between node, between socket, or even between core, for optimal results, depends on the architecture and communication speeds. The shared memory parallelism makes sure each hardware core is used the best way possible while sharing a certain memory level with its neighbouring cores through threads.

The communication profile of the hybrid code involves lesser messages between MPI processes, with the messages being larger in general. This reduced communication is the main driver of performance improvements in hybrid code [5]. A decrease in memory use is another advantage of the hybrid codes [20].

**Implementation**

The program does Direct Numerical Simulation, with a given inlet velocity, lateral and top and bottom periodic boundaries and an outlet condition designed to allow for smooth exit of turbulent structures and no reflections back to the computation domain. This allows for the simulation of a spatial, or Eulerian, turbulent planar jet.

The non streamwise directions are discretized with a pseudo-spectral scheme and the streamwise direction uses a compact finite differences scheme. The time advance is accomplished using a 3rd order accurate Runge Kutta explicit scheme.

This program is hybridly parallelized with MPI and OpenMP.

**Spatial Discretization**

To make the transforms from physical to Fourier space, a Discrete Fourier Transform is applied. The general definition of the direct Fourier transform is given by equation

$$\hat{φ}(x, k_y, k_z, t) = \frac{1}{n_y n_z} \sum_{j=-n_y/2}^{n_y/2-1} \sum_{k=-n_z/2}^{n_z/2-1} \phi(x, y, z, t) e^{ik_y y + ik_z z} \tag{4}$$

While the inverse Fourier transform to go back to physical space is given by

$$\phi(x, y, z, t) = \sum_{j=-n_y/2}^{n_y/2-1} \sum_{k=-n_z/2}^{n_z/2-1} \hat{φ}(x, k_y, k_z, t) e^{ik_y y + ik_z z} \tag{5}$$

The implemented discrete transforms use the Fast Fourier Transform of [9], designated FFTW (Fastest Fourier Transform in the West).

For the same resolution, pseudo-spectral schemes have a computational cost proportional to \(n \log_2 n\), with \(n\) the number of discretization points, this is a very low computational cost compared to other schemes that commonly have \(n^2\) proportional costs [14].

To differentiate in any of the pseudo-spectral directions while in the spectral space, a multiplication by \(-ik_i\) is needed, yielding an exact derivative.

$$\frac{∂\hat{φ}}{∂y} = -i k_y \hat{φ} \tag{6}$$

$$\frac{∂\hat{φ}}{∂z} = -i k_z \hat{φ} \tag{7}$$

Therefore, spectral schemes’ differentiation is not only exact, as it is very light computationally, consisting of simple multiplications in the Fourier space, with the accuracy only suffering from the rounding errors and the discretization for the Fourier Transform [8].

Multiplications on the other hand become convolution products in the Fourier space, which are complex operations, for this reason, products in the algorithm are performed in the physical space.

For the streamwise direction there exists no periodicity in the boundaries. To attain the desired order of accuracy compact finite difference schemes are used. The 6th order compact finite differences scheme described in [10] is used.

$$\sum_{j=-p}^{q} \alpha_j f_{i+j} = \sum_{k=-q}^{q} \alpha_k f_{i+j} + \mathcal{O}(\Delta x^n) \tag{8}$$

Compact finite difference schemes express the derivative in a given point as a function of its neighbouring points value and derivative. For each iteration that requires a derivative in the streamwise direction, a number of \(n_y n_z\) systems of \(n_x\) equations has to be solved, and another equal number of equations for the second derivative.

For the first derivative this results in

$$\alpha_j f'_{i-1} + \alpha_j f'_{i+1} = a f'_{i+1} - f'_{i-1} + b f'_{i+2} - f'_{i-2} \frac{Δx}{Δx} \tag{9}$$

and for the second derivative in

$$\alpha_j f''_{i-1} + \alpha_j f''_{i+1} = a f''_{i+1} - 2 f''_{i-1} + f''_{i-2} + b f''_{i+2} - 2 f''_{i-1} + f''_{i-2} \frac{Δx^2}{Δx^2} \tag{10}$$
For the points near the inlet and outlet boundaries the 4th order Padé scheme is applied, and for the boundaries a 3rd order implicit scheme is used. The solving of these system can be written in matrix form, and the computation method is is shown in equation 11.

\[ f' = A_1^{-1} \left( \frac{1}{\Delta x^2} B_1 f \right); f'' = A_2^{-1} \left( \frac{1}{\Delta x^2} B_2 f \right) \] (11)

Temporal Discretization
The time advance method is a three fractional step, 3rd order accurate explicit Runge-Kutta time integration scheme [13] [12].
To apply this method, the right hand side of the momentum equation is split into terms that are calculated separately.

\[ \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = \frac{1}{\rho} \frac{\partial P}{\partial x_j} - P \delta_{ij} + \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \] (12)

If the mass conservation is accounted for the momentum equation becomes

\[ \frac{\partial u_i}{\partial t} = -u_j \frac{\partial u_i}{\partial x_j} - \frac{1}{\rho} \frac{\partial P}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \] (13)

Therefore the momentum equation can be written in the form

\[ \frac{\partial u_i}{\partial t} = N(u_i) + L(u_i) - \frac{\partial \pi^p}{\partial x_i} \] (14)

Where the Non-Linear term is

\[ N(u_i) = -u_j \frac{\partial u_i}{\partial x_j} + \frac{\partial P}{\partial x_i} u_j - u_j \frac{\partial u_i}{\partial x_j} + u_j \frac{\partial u_j}{\partial x_i} = u_j \times \omega_i \] (15)

the Linear term is

\[ L(u_i) = \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \] (16)

and the Modified Pressure is

\[ \pi^p = \frac{P}{\rho} + \frac{u_j u_j}{2} \] (17)

With these terms calculated, the time step is possible by applying

\[ \frac{u_i^p - u_i^{p-1}}{\Delta t} = \alpha_p \left[ N(u_i^{p-1}) + L(u_i^{p-1}) \right] + \beta_p \left[ N(u_i^{p-2}) + L(u_i^{p-2}) \right] - \frac{\partial \pi^p}{\partial x_i} \] (18)

The pressure term is unknown. To calculate the pressure term a temporary velocity field is defined that does not comply with mass conservation yet.

\[ \frac{u_i^p - u_i^{p-1}}{\Delta t} = \alpha_p \left[ N(u_i^{p-1}) + L(u_i^{p-1}) \right] + \beta_p \left[ N(u_i^{p-2}) + L(u_i^{p-2}) \right] \] (19)

Computationally this achieved with

\[ \frac{u_i^p - u_i^{p-1}}{\Delta t} = \alpha_p \left[ \hat{N}_i^{p-1} + \nu \left( \frac{d^2}{dx^2} - k^2 \right) u_i^{p-1} \right] + \beta_p \left[ \hat{N}_i^{p-2} + \nu \left( \frac{d^2}{dx^2} - k^2 \right) u_i^{p-2} \right] \] (20)

Subtracting equation (equation 19) to the original equation (equation 18), by applying the divergence operator, and enforcing the mass conservation on the final velocity field, this results in equation 21.

\[ \nabla \cdot u_i^p = -\nabla^2 \pi^p \] (21)

This is a Poisson equation that is used to compute the pressure that is used to correct the temporary velocity field, yielding the final correct velocity field through equation 22.

\[ u_i^p = u_i^{p-1} - \Delta t \frac{\partial \pi^p}{\partial x_i} \] (22)

Stability
Numerical stability is ensured by enforcing a maximum for the CFL number.

\[ \Delta t = CFL \times \min \left\{ \frac{\Delta x}{|u_z|_{max}}, \frac{\Delta y}{\pi |u_y|_{max}}, \frac{\Delta z}{\pi |u_z|_{max}} \right\} \] (23)

Boundary conditions
For the inlet boundary, the conditions to realistically simulate a jet were implemented. This is done by superimposing a noise profile at the shear layer area to the mean normal velocity profile that correctly simulates a planar jet.

The side and top and bottom boundaries are periodic due to the spectral scheme. For correct implementation these need to be far away from the
turbulent area so perturbations from one side of the domain do not affect the opposite side.

The outlet boundary uses a non-reflective outflow boundary condition \[ [18] \], that advances both the convective and diffusive terms of the momentum equation.

Parallelization and HPC implementation of the Algorithm

This work is based on the work previously done in \[ [7] \] and later \[ [14] \], that used a pure MPI parallelization, and said to have as performance limiting factors the communication time and memory use.

The data dependency analysis done by \[ [7] \] resulted in the current mixed domain decomposition. With parallelization for the FFT in slabs perpendicular to the Jet/Compact Scheme and parallelization for the Compact Scheme in slices perpendicular to the FFT slabs.

Hybrid parallelization

To make the hybrid parallelization, OpenMP for shared memory parallelization was retrofitted to the existing code, in compatibility with the MPI parallelization, for two parallelism levels.

The program is written in the FORTRAN, in the form of the FORTRAN 95 standard \[ [1] , [16] \].

Hybrid subdomain transposition

To be able to use pseudo-spectral methods in two directions and a compact FD scheme in the third direction, a subdomain transposition is required to ensure both the pseudospectral scheme and the compact finite difference scheme have all the necessary data points for their operation. This transposition demands an MPI\textsc{AllToAll} communication intensive step at every subdomain transposition.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{subdomain_transposition.png}
\caption{Subdomain transposition by MPI\textsc{AllToAll} call. \[ [14] \]}
\end{figure}

For a Pure MPI implementation, the volume of an MPI \textsc{AllToAll} transfer will be proportional to

\[ \text{Volume}_{\text{MPI \textsc{Transfer}}} \propto N_{\text{Nodes}}^2 \text{MPI} \] \hspace{1cm} (24)

And an Hybrid application, the volume of an MPI\textsc{AllToAll} communications will be proportional to

\[ \text{Volume}_{\text{Hybrid Transfer}} \propto N_{\text{Nodes}}^2 \text{MPI} \cdot \text{Threads}_{\text{per Node}} \] \hspace{1cm} (25)

Therefore, if it assumed that each of the threads inside each node were being treated as a separate node before a code is hybridized from MPI to MPI+OpenMP, and neglecting the overheads of OpenMP initialization and different communication speeds between internode and intranode communications, the theoretical speedup in communications will be proportional to the number of threads in each node, resulting in equation 26.

\[ \text{Volume}_{\text{Hybrid Transfer}} \propto \frac{N_{\text{Nodes}}^2 \text{PureMPI}}{\text{Threads}_{\text{per Node}}} \] \hspace{1cm} (26)

Hybrid model

The approach taken with respect to overlapping communication or not, is the MPI THREAD FUNNELED option, in a master only mode, where the master thread only communicates outside of OpenMP parallel regions \[ [21] \]. It does not allow simultaneous processing and communication between threads and it forces the non-master threads of each MPI to idle while the communication takes place. But since the code bottleneck is the communications step and since the communication happens with an MPI\textsc{AllToAll} that must necessarily be done and does not overlap computing, the gains to be obtained by taking a more complicated approach were weighted against the increased programming of an alternative approach.

In this model, the performance is limited by one of two situations \[ [19] \]. If the master thread saturates the communication capacity of the node, this saturation will be the performance limiting factor. If the master thread does not saturate the interconnect of the node, the performance will be limited by the idling threads and the pure MPI version will have a better performance.

It can be concluded that the critical factor determining which of the problems happens is interconnect speed. Using a fast interconnect, the pure MPI code will outperform the hybrid up to a larger number of cores than with a slow interconnect \[ [5] \] and a slow connection system will benefit the most from an hybrid implementation \[ [11] \].

Single forking vs. Multiple forking

The approach taken was to have multiple forking and joins along the code. This approach was decided based on the fact that the overheads were of negligible importance for fine meshes and more amenable implementation effort.
The FFTW library was parallelized manually, being executed by the program developed, calling the FFTW library the required number of times from each OpenMP thread separately, in an explicit way.

The work of [23] did not show "any significant differences in performance between implicit and explicit" implementations of threaded FFTW3.

Packing and unpacking subroutines
In the subdomain transposition step, the transposition subroutines have three steps, packing the send buffer, communicating, and unpacking the received buffer. The functioning of these routines had to be changed to make the OpenMP parallel.

Thread safety
With respect to subroutines calls there was the option to parallelize the called subroutines or to parallelize the call to the subroutines, this second option demands that the called subroutine is thread safe. According to the number of calls to each subroutine and consequently the overheads, some subroutines were made thread safe while other were parallelized.

Other optimizations
Other optimizations studied and implemented were the loop reordering due to the change of behaviour of certain subroutines when hybridized, which allowed for better use of the memory arrangement of FORTRAN; and the inclusion of different inner loops that shared the same outer loop, avoiding repeated loops. These changes were studied and implemented according to the results.

Loop scheduling
Loop scheduling, which defines the grain of the parallelization, was set to runtime so tests could be run with different scheduling.

Profiling
The detailed profiling was done using the TAU tool [22].

Results
Verification was done. Profiling was done for the pure MPI version. Specific implemented details and parts were tested. Hybrid performance was tested. And profiling of the hybrid version was done.

Verification
Comparative runs with the previous, physics validated, version of the code were done. This was done both in serial mode and in multicore multinode machines. The program gave equal results, to machine precision, in every comparative run to the physics validated version.

An illustrative simulation of a Newtonian fluid planar jet with 75 million points was done with a $Re = 4500$ with 16 MPI processes with 2 OpenMP threads each. Vorticity is plotted.

Profiling of Pure MPI version
Profiling was done of the pure MPI version was done a consumer laptop with an Intel i7-4720HQ processor and 8 GB RAM memory running Linux.

The simulation used a $512 \times 512 \times 256$ mesh, $Re = 100$ and 26 iterations with IO at every 5 timesteps.

The longest time consuming subroutines in terms of time spent running them exclusively (not including call made the subroutines themselves) is plotted.

Fraction of program that is OpenMP parallelizable
The top 10 most time consuming subroutines make $\approx 92\%$ of the time. If the MPI_ALLTOALL and MPI_File_sync subroutines, which do not need to be repeated in the same node, are discounted (13.2%), the remaining 8 most time consuming subroutines make up to $79/(100 - 13.2) = 91\%$ of the work, and are parallelizable with OpenMP.

Analysis of Hybrid loops
Some tests were done to specific parts of the code to take decisions related with the implementation.
A single loop was tested for speedup. No great improvement in time was measured, with average time per loop going from 20.2s with 2 MPI processes to 17.8s with 2 MPI processes with 2 OpenMP threads each.

Tests were done for static and dynamic scheduling of loops. The results were inconclusive showing marginal differences between the two cases.

Nested loops hosted in similar separate outer loops were joint and the times of each implementation were compared.

Table 1: Inclusive times for joint and separate external loop

<table>
<thead>
<tr>
<th>Node</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Mean</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17918</td>
<td>17919</td>
<td>17887</td>
<td>71549</td>
</tr>
<tr>
<td>1</td>
<td>17956</td>
<td>17755</td>
<td>17887</td>
<td>71549</td>
</tr>
<tr>
<td>Mean</td>
<td>17887</td>
<td>17887</td>
<td>17887</td>
<td>71549</td>
</tr>
<tr>
<td>Total</td>
<td>35745</td>
<td>35745</td>
<td>35745</td>
<td>143098</td>
</tr>
</tbody>
</table>

An improvement in performance in the order of 10% was observed.

Packing and unpacking subroutines were also tested. No improvements in performance were measured.

Scalability of OpenMP parallelization
Test were run to access the success of the parallelization with OpenMP. On the specific case of 1 MPI process with 1 OpenMP thread or 8 OpenMP threads, the parallelized fraction is calculated at 75%.

The results are plotted in figure 4.

Figure 4: OpenMP times as function of number of OpenMP threads, normalized to time with 1 OpenMP thread. For various numbers of MPI processes.

Scalability of hybrid parallelization
With respect to the hybrid performance, a test simulation was ran with different combinations of MPI processes and OpenMP threads to access what was the most effective combination on LASEFs cluster galego.

Due to a communication hardware problem between nodes in galego, programs could only be ran on one node at a time, therefore for these performance comparisons used only one node.

Table 2: Hybrid scalability test

<table>
<thead>
<tr>
<th>Number of MPI processes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time of simulation in seconds</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>159</td>
<td>158</td>
<td>154</td>
<td>145</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>375</td>
<td>286</td>
<td>217</td>
<td>-</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>269</td>
<td>228</td>
<td>204</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>188</td>
<td>194</td>
<td>217</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>210</td>
<td>175</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>260</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

These results favor the hybrid implementation compared to a pure MPI implementation for the tested architecture. The best performance of the hybrid model was of 102s of run time in the $16_{MPI} \times 2_{OpenMP}$ case, compared with 137s of run time of the pure MPI model with 32 processes, offering a 34% increase in performance.

The hybrid model with $4_{MPI} \times 4_{OpenMP}$ also attained a performance similar (138s) to the best pure MPI performance (137s) with 32 $MPI$, using half of the resources.

Even though the Hybrid MPI+OpenMP version does not always attain the performance of the pure MPI implementation on the same node, when testing in a multinode architecture, the communication...
reduction should offer advantage to the hybrid implementation, and even in lower performance cases, the hybrid version will always use less memory, specially in cases where central finite differences schemes are applied. This proves the concept of a hybrid DNS spatial simulation program, and shows it has potential further advantages to run simulations that employ passive scalars, or FENE-P models of viscosity using centered finite difference schemes.

Profiling of hybrid version
The run used a mesh of $768 \times 768 \times 128$ points, doing 151 iterations, with IO steps at every 50 iterations, with 4 MPI processes and 4 OpenMP threads each. The profiling shows a correct implementation of the hybrid model.

Even in this simulation that ran on a single node, which should have very low communication times between MPI processes in itself, the MPI ALLTOALL subroutine is taking 25% of the program runtime.

The tridg9 subroutine, in dark grey on figure 6, not executed in by the 4th node, shows the inherent load unbalance of the MPI parallelization due to the dealiasing scheme used.

Concluding remarks
This work analyzed different parallelization models and other methods to increase the performance of a direct numerical simulation program in a high performance computing setting.

First, a detailed study of an algorithm for direct numerical simulations was done. This algorithm uses explicit time advance and is pseudo spectral in two directions and uses compact FD in one direction, and is used to simulate a turbulent plane jet. Second, current trends in the programming and high performance computing communities were studied, and possible approaches were laid out. Third, a detailed study of the performance of the simulation algorithm was done, to identify hotspots and bottlenecks. Fourth, the applicable approaches to the simulation algorithm were selected, implemented and tested, to determine the best approach to follow. Lastly overall performances were compared.

From the literature review and the DNS method used, it can be concluded that a parallelization with communication and computation overlap would require an extreme programming effort, with results obtained showing that the gains to be harvested with the used approach are significant. This leads to the conclusion that in this case a funneled or master only hybrid parallelization yields the best results-to-resources relation possible.

A possible conclusion of this work is that an hybrid parallelization will only yield performance gains with a very high resource investment, not only of hardware, but also of programming resources.

It can also be seen that the subdomain changes that happen along the program run, take a large portion of the total time. This asks for the consideration of a different algorithm for this type of simulation.

The effects of different granularity of the fine grain level parallelization were shown to have a non significant effect.

The optimization of joining nested loops showed a small increment in performance.

The hybrid MPI+OpenMP implementation showed improvements, including the best performance, with 34% improvement over pure MPI, and performance as good as the best pure MPI case with half the computational resources. It also showed lower performance for some cases when...
compared with the pure MPI implementation. Despite this, no comparative performance tests were completed on multinode machines, and it is in that case that the hybrid version should show its superior performance, by avoiding many unnecessary slow communications. Furthermore, memory use gains will be present which allow for larger, even if not significantly faster, simulations. This can be a decisive factor to choose the hybrid model when passive scalar or alternative viscosity models which use centered finite difference schemes are used. These three facts show promising results for the hybrid MPI+OpenMP parallelization model applied to a DNS simulation.

Achievements
This work involved tasks that were the first of their type at LASEF. It is the first time a high detail profiling of LASEF’s spatial jet direct numerical simulation program was done and hotspots and bottlenecks were identified. Previous analyses of the in-house developed software were done through scarce manual instrumentation.

An OpenMP parallelization was applied to a spatial direct numerical simulation program. An hybrid parallelization was thus achieved with the shared memory OpenMP parallelization retrofitted on the distributed memory MPI parallelization previously implemented. This was the first instance of a hybrid parallel DNS program at LASEF, and in this process, the first hybrid parallel program developed at LASEF.

Code verification was done and a 75.5 million point simulation (768 × 768 × 128 mesh) of 10000 timesteps was done to demonstrate the program and prove the hybrid parallelization concept.

Performance analysis and a detailed profiling was done for the developed Hybrid MPI+OpenMP code.

Performance comparisons were made.

The best performance of the spatial jet DNS was attained by the hybrid model implemented, and similar performance as the previously fastest version was attained with less computational resources.

Future Work
Future could involve the following items.

• Implement heterogeneous domain decomposition for better MPI load balance.

• Single forking/joining for minimum OpenMP overheads.

• Passive Scalar, LES, and FENE-P models implementation to take advantage of better memory use.

• Study hybrid parallel IO.

Acknowledgements
The author would like to thank LASEF for the facilities to conduct this work.

References


