# Implementation of a Harmonic Balance Solver into a Compressible CFD Code

Technical report 2017:05

# Implementation of a Harmonic Balance Solver into a Compressible CFD Code

by

DANIEL LINDBLAD

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2017

# Implementation of a Harmonic Balance Solver into a Compressible CFD Code

D A N I E L   L I N D B L A D

Technical report 2017:05
ISSN 1652-8565

Division of Fluid Dynamics
Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg
Sverige
Telephone +46 (0)31 772 1000

# Implementation of a Harmonic Balance Solver into a Compressible CFD Code

Introduction to High Performance Computing 2016

*Author:*
Daniel Lindblad
daniel.lindblad@chalmers.se

October 19, 2017

## 1 Introduction/Abstract

In this project the Harmonic Balance technique for solving time periodic problems in fluid dynamics has been implemented into the parallel Navier-Stokes solver G3D::Flow. Performance issues with regards to the original implementation are identified using two profiling tools, Cachegrind [1.] and Allinea-Map 6.11 [2.]. These findings are used to derive a numerical algorithm that benefits from both improved cache locality and loop performance by utlizing blocked matrix multiplication and vectorization. The scaling properties of the Harmonic Balance technique are also demonstrated by deriving an approximate performance model and comparing it to numerical results obtained at the Triolith and Hebbe computer clusters at Linköping and Chalmers Univeristy.

## 2 Harmonic Balance Method

### 2.1 Governing Equations

The Harmonic Balance technique is applicable to problems in which the solution to the governing equations is periodic in time. For some situations in fluid dynamics this holds true, in which case the flow can be described by the compressible Navier-Stokes equations

$$\frac{\partial \mathcal{Q}}{\partial t} + \frac{\partial \mathcal{F}_j}{\partial x_j} = 0 \tag{1}$$

The state vector $\mathcal{Q} = (\rho, \rho u, \rho v, \rho w, \rho e_0)^T$ contains the conserved variables (density, momentum and total energy) and $\mathcal{F}_j$ is the total flux vector. The Harmonic Balance equations are derived based on the assumption that the conserved variables in $\mathcal{Q}$ are periodic in time with a known period $T$. This assumption implies that the solution can be expressed as a Fourier series in time, with spatially varying coefficients

$$\mathcal{Q}(t, x_j) = \sum_{n=-\infty}^{\infty} \hat{\mathcal{Q}}_n(x_j) e^{i\omega_n t} \tag{2}$$

Here, $\omega_n = 2\pi n/T$ is the angular frequency of the $n^{th}$ harmonic. The Fourier series can be truncated if we assume that the flow can be accurately described by a limited number of harmonics

$$\mathcal{Q}(t) \approx \sum_{n=-N_h}^{N_h} \hat{\mathcal{Q}}_n e^{i\omega_n t} \tag{3}$$

The aim of the Harmonic Balance technique is to recast the problem of finding the evolution of $\mathcal{Q}(t)$ by using time stepping techniques to finding the amplitude of the harmonic amplitudes $\hat{\mathcal{Q}}_n$. However, instead of directly formulating governing equations for the $2N_h + 1$ unknown harmonic amplitudes the solution is sought at $N_t = 2N_h + 1$ time instances equally distributed over a period. These time instances, or time levels, represent frozen snapshots of the transient solution at times $t_l = lT/N_t$ and are grouped together in a new state vector

$$\mathcal{Q}^* = \begin{bmatrix} \rho_0 \\ \rho_1 \\ \vdots \\ \rho_{N_t-1} \\ \rho u_0 \\ \vdots \\ \rho u_{N_t-1} \\ \vdots \\ \rho e_{0,0} \\ \vdots \\ \rho e_{0,N_t-1} \end{bmatrix} \tag{4}$$

We will now derive a governing equation for $\mathcal{Q}^*$ and show how the harmonic amplitudes can be computed from the new state vector. First, the Fourier series assumption in Eq. (3) is substituted into the time derivative of the Navier-Stokes equations

$$\frac{\partial \mathcal{Q}}{\partial t} \approx \sum_{n=-N_h}^{N_h} i\omega_n \hat{\mathcal{Q}}_n e^{i\omega_n t} \tag{5}$$

The harmonic amplitudes are furthermore computed with a discrete Fourier transorm over all time levels

$$\hat{\mathcal{Q}}_n = \frac{1}{N_t} \sum_{l=0}^{N_t-1} \mathcal{Q}_l \; e^{-i\omega_n t_l} \tag{6}$$

Here, $\mathcal{Q}_l = \mathcal{Q}(t_l)$ represents the solution at time level $l$. If Eq. (6) is inserted into Eq. (5) and the resulting expression is evaluated at time level $m$, the following relation is obtained after some algebra

$$\frac{\partial \mathcal{Q}_m}{\partial t} \approx \sum_{l=0}^{N_t-1} \left( \frac{i}{N_t} \sum_{n=-N_h}^{N_h} \omega_n e^{i\omega_n (t_m - t_l)} \right) \mathcal{Q}_l \tag{7}$$

In this equation we have obtained a high-order finite difference approximation of the time derivative at time level $m$ by means of the solution at all other time levels. This can also be expressed in terms of a matrix multiplication according to

$$\frac{\partial}{\partial t} \mathcal{Q}^* \approx D \mathcal{Q}^* \tag{8}$$

The matrix $D$ is referred to as the time spectral derivative matrix. It takes on the following form

$$D = \begin{bmatrix} [d_{m,l}] & & & \mathbf{0} \\ & [d_{m,l}] & & \\ & & \ddots & \\ \mathbf{0} & & & [d_{m,l}] \end{bmatrix} \tag{9}$$

Here, $[d_{m,l}]$ is an $N_t \times N_t$ matrix whose elements are given by the expression inside the large parenthesis of Eq. (7). The number of diagonal blocks in $D$ corresponds to the number of variables in $\mathcal{Q}$, which in our case is $N_{Var} = 5$. Thus the size of $D$ is $(N_{Var} \cdot N_t) \times (N_{Var} \cdot N_t)$. A governing equation for $\mathcal{Q}^*$ can now be formulated by replacing the time derivative by the matrix multiplication in Eq. (8)

$$D\mathcal{Q}^* + \frac{\partial \mathcal{F}_j^*}{\partial x_j} = 0 \tag{10}$$

This represents a coupled set of mathematically steady state equations for the solution at each time level. The new flux vector $\mathcal{F}_j^*$ furthermore contains the flux for each variable and time level. Equation (10) is discretized in space using the finite volume method and a pseudo time derivative is introduced to drive the resulting system of equations towards steady state using local time stepping

$$V_i \frac{\partial \mathcal{Q}_i^*}{\partial \tau} + V_i D \mathcal{Q}_i^* + \mathcal{R}_i^* = 0 \tag{11}$$

In this equation index $i$ represents a specific cell in the computational mesh. The flow residual $\mathcal{R}_i^*$ furthermore contains the discretized flux for each variable and time level in cell $i$

$$\mathcal{R}_i^* = \begin{bmatrix} \mathcal{R}_0^{\rho} \\ \mathcal{R}_1^{\rho} \\ \vdots \\ \mathcal{R}_{N_t-1}^{\rho} \\ \mathcal{R}_0^{\rho u} \\ \vdots \\ \mathcal{R}_{N_t-1}^{\rho u} \\ \vdots \\ \mathcal{R}_0^{\rho e_0} \\ \vdots \\ \mathcal{R}_{N_t-1}^{\rho e_0} \end{bmatrix}_i \tag{12}$$

It should be noted that the residual flux for each variable and time level can be computed from the solution at that time level, for example $\mathcal{R}_l^{\rho} = \mathcal{R}^{\rho}(\mathcal{Q}_l)$, except when special boundary conditions are used in the simulation.

## 2.2 Integration into G3D::Flow

The Harmonic Balance method was implemented into Chalmers' in-house CFD solver G3D::Flow. The code uses distributed datatypes in the PETSc library together with domain decomposition to run on multiple processes. In the original solver a three stage Runge-Kutta cycle is used to update the solution in time

$$
\begin{aligned}
\mathcal{Q}_i^{\mathrm{a}} &= \mathcal{Q}_i^n - \frac{\Delta t}{V_i}\mathcal{R}_i^n \\
\mathcal{Q}_i^{\mathrm{b}} &= \frac{1}{2}(\mathcal{Q}_i^{\mathrm{a}} + \mathcal{Q}_i^n) - \frac{\Delta t}{2V_i}\mathcal{R}_i^{\mathrm{a}} \\
\mathcal{Q}_i^{n+1} &= \frac{1}{2}(\mathcal{Q}_i^{\mathrm{a}} + \mathcal{Q}_i^n) - \frac{\Delta t}{2V_i}\mathcal{R}_i^{\mathrm{b}} \qquad (13)
\end{aligned}
$$

Superscript a and b denotes intermediate solutions in the Runge-Kutta cycle and $n$ refers to the time step. In the Harmonic Balance solver the solution is instead updated in pseudo time $\tau$ with a time step that is adjusted individually for each cell. A converged solution is obtained when the solution does not change from time step to time step, i.e. when the flow residual and time spectral derivative balances in Eq. (11). The Runge Kutta cycle may therefore be formulated as follows

$$
\begin{aligned}
\mathcal{Q}_i^{*,\mathrm{a}} &= \mathcal{Q}_i^{*,n} - \left(\frac{\Delta\tau_i}{V_i}\mathcal{R}_i^{*,n} + \Delta\tau_i D\mathcal{Q}_i^{*,n}\right) \\
\mathcal{Q}_i^{*,\mathrm{b}} &= \frac{1}{2}(\mathcal{Q}_i^{*,\mathrm{a}} + \mathcal{Q}_i^{*,n}) - \frac{1}{2}\left(\frac{\Delta\tau_i}{V_i}\mathcal{R}_i^{*,\mathrm{a}} + \Delta\tau_i D\mathcal{Q}_i^{*,\mathrm{a}}\right) \\
\mathcal{Q}_i^{*,n+1} &= \frac{1}{2}(\mathcal{Q}_i^{*,\mathrm{a}} + \mathcal{Q}_i^{*,n}) - \frac{1}{2}\underbrace{\left(\frac{\Delta\tau_i}{V_i}\mathcal{R}_i^{*,\mathrm{b}} + \Delta\tau_i D\mathcal{Q}_i^{*,\mathrm{b}}\right)}_{\tilde{\mathcal{R}}_i} \qquad (14)
\end{aligned}
$$

The same type of domain decomposition that is used for the original solver is kept for the Harmonic Balance implementation. This implies that each process stores all time levels of the solution in the cells that it handles. There is thus no attempt to parallelize the code further with respect to the time levels, altough it is potentially possible. The flow field and residual are stored in two arrays of length $N_t$, where each entry contains one instance of the same classes that are used to store flow fields and residuals in the original solver. Each of these objects does in turn contain $N_{Var}$ arrays of type PetscScalar (similar to double) which stores either the flow field variables or residual for all cells ($N_{CellsLocal}$) that the process owns. The data format is summarized below for the flow field variables

$$
\mathcal{Q}^* = \begin{bmatrix}
\mathcal{Q}_0 : & \begin{bmatrix} \text{PetscScalar} & \rho[N_{CellsLocal}] \\ \text{PetscScalar} & \rho u[N_{CellsLocal}] \\ \text{PetscScalar} & \rho v[N_{CellsLocal}] \\ \text{PetscScalar} & \rho w[N_{CellsLocal}] \\ \text{PetscScalar} & \rho e_0[N_{CellsLocal}] \end{bmatrix} \\
\mathcal{Q}_1 : & \begin{bmatrix} \text{PetscScalar} & \rho[N_{CellsLocal}] \\ \text{PetscScalar} & \rho u[N_{CellsLocal}] \\ \text{PetscScalar} & \rho v[N_{CellsLocal}] \\ \text{PetscScalar} & \rho w[N_{CellsLocal}] \\ \text{PetscScalar} & \rho e_0[N_{CellsLocal}] \end{bmatrix} \\
 & \vdots \\
\mathcal{Q}_{N_t-1} : & \begin{bmatrix} \text{PetscScalar} & \rho[N_{CellsLocal}] \\ \text{PetscScalar} & \rho u[N_{CellsLocal}] \\ \text{PetscScalar} & \rho v[N_{CellsLocal}] \\ \text{PetscScalar} & \rho w[N_{CellsLocal}] \\ \text{PetscScalar} & \rho e_0[N_{CellsLocal}] \end{bmatrix}
\end{bmatrix} \tag{15}
$$

Note that this storage format is slightly different from how the new state vector was presented in Eq. (4). This is however merely a cosmetic difference, since in reality the $\mathcal{Q}^*$ and $\mathcal{Q}_l$ objects in Eq. (15) are merely placeholders for the vectors that store the actual data ($\rho[]$ ...). In terms of memory layout the code does therefore only see $N_t \cdot N_{Var}$ arrays in which the conserved variables and corresponding residuals are stored for each cell. A different way to see this is that the code stores the flow field in terms of a matrix of size $(N_t \cdot N_{Var}) \times N_{CellsLocal}$ which has row-major order data storage. This insight will later be used to derive an efficient algorithm for computing $D\mathcal{Q}^*$ but first the skeleton of the Harmonic Balance Runge-Kutta implementation is presented below in pseudo code

```
/* Runge Kutta Stage 1 */

// Update Ghost Cells
MPI_Scatter();

// Update Flow Field Residual for all time levels
for(int it=0; it<Nt; it++){

   // Backup Flow Field
   Cons0[it] = Cons[it];

   // Update Flow Residual for all variables at time level it
   // Res = (dt/V)*R
   Res[it]->update();

}

// Add Time Spectral Derivative to Flow Field Residual
ExplicitSpectralDerivative();
```

```
// Update Solution for RK Stage 1
for(int it=0; it<Nt; it++){

  // Res[it] includes D*Q
  Cons[it] = Cons0[it] - Res[it];

}

/* Runge Kutta Stage 2 */

// Update Ghost Cells
MPI_Scatter();

// Update Flow Field Residual for all time levels
for(int it=0; it<Nt; it++){

  // Backup Flow Field
  Cons0[it] = 0.5*(Cons[it] + Cons0[it]);

  // Update Flow Residual for all variables at time level it
  // Res = (dt/V)*R
  Res[it]->update();

}

// Add Time Spectral Derivative to Flow Field Residual
ExplicitSpectralDerivative();

// Update Solution for RK Stage 2
for(int it=0; it<Nt; it++){

  // Res[it] includes D*Q
  Cons[it] = Cons0[it] - 0.5*Res[it];

}

/* Runge Kutta Stage 3 */

// Update Ghost Cells
MPI_Scatter();

// Update Flow Field Residual for all time levels
for(int it=0; it<Nt; it++){

  // Update Flow Residual for all variables at time level it
  // Res = (dt/V)*R
  Res[it]->update();
}

// Add Time Spectral Derivative to Flow Field Residual
ExplicitSpectralDerivative();

// Update Solution for RK Stage 3
for(int it=0; it<Nt; it++){

  // Res[it] includes D*Q
  Cons[it] = Cons0[it] - 0.5*Res[it];

}
```

---

The variables "Res", "Cons" and "Cons0" respectively contain the total residual ($\tilde{\mathcal{R}}^*$), flow field ($\mathcal{Q}^*$) and a backup of the flow field necessary for the Runke Kutta cycle. They all have the storage format presented in Eq. (15), i.e. Cons[it] $= \mathcal{Q}_{\text{it}}$.

Each Runge Kutta stage begins by scattering ghost cell values between all process so that the cell values at the boundaries of each sub-domain are updated. After this the code goes through each time level and adds the flow residual $((\Delta\tau_i/V_i)\mathcal{R}_i^*)$ to "Res" by calling the G3D::Flow flux routines (Res[it]->update();). Finally the spectral derivative matrix $(\Delta\tau_i D\mathcal{Q}_i^*)$ is added to "Res" inside the routine called "ExplicitSpectralDerivative()". The initial implementation of this routine is presented in the next section.

## 2.3 Original Time Spectral Derivative

The first implementation of the time spectral derivative was done in a straightforward manner to allow easy debugging and verification of the code. Note that for each cell the matrix multiplication presented in Eq. (8) can be split up in $N_{Var}$ smaller matrix multiplications due to the block structure of $D$. This is done in a straightforward way according to

---

```
void ExplicitSpectralDerivative()
{
  // Loop over all cells owned by process
  for(int ic=0; ic<NCellsLocal; ic++){
        // Loop through all variables (blocks of D)
    for(int ivar=0; ivar<NVar; ivar++){
      // Loop through all residuals
      for(int m=0; m<Nt; m++){
        // Loop through all time levels
        for(int l=0; l<Nt; l++){
          Res[m]->Var[ivar][ic] += dtime[ic]*d[m,l]* ...
                                   Cons[l]->Var[ivar][ic];
        }
```

```
         }
      }
   }
   return;
}
```

In this presudo code, "Res[m]->Var[ivar]" points to the first element of the residual array for variable "ivar" at time level "m", see Eq. (15). Also note that the code stores the local time step for a given cell using the array "dtime".

## 2.4 Approximate Performance Model for Harmonic Balance

The execution time of one Runge Kutta cycle was estimated based on the number of operations that are necessary to update the ghost cells, flow residual and time spectral derivative. The addition and assignment operations that are used to update "Cons" and "Cons0" are also taken into consideration.

If we assume that the computational domain contains $N_{Cells}$ cells that are split between $N_{Proc}$ processors the number of local cells become $N_{CellsLocal} = N_{Cells}/N_{Proc}$. G3D::Flow operate on hexahedral, block-structured meshes in which each block contains $N_i \cdot N_j \cdot N_k$ cells. For simplicity we assume that the computational domain is made up of one block with equal number of cells in each block direction ($N_i = N_j = N_k$). In addition, we assume that the domain decomposition is performed by splitting the block along one edge, so that each sub domain consist of $N_i N_j N_k / N_{Proc}$ cells. In this case we see that the number of ghost cells that each process must perform MPI communication over becomes proportional to $2N_{Cells}^{2/3}$, one cell layer above and one below. For a more general domain topology this relation only becomes an estimate but can still give an indication on the size of the MPI communication. For each cell where MPI communication is performed, $N_t \cdot N_{Var}$ variables must be communicated, implying that the total complexity of updating the ghots cells becomes proportional to $2N_{Cells}^{2/3} N_{Var} N_t$.

Updating the flow residual in finite volume methods implies calculating the flux over all faces in the computational mesh and add the net contribution to the cells connected to that face. For a hexahedral mesh the number of faces each process must take care of becomes $3N_{Cells}/N_{Proc}$. The flow residual must be updated for all variables and time levels, giving a total complexity proportional to $3N_{Cells}N_{Var}N_t/N_{Proc}$.

The operations used to update "Cons" and "Cons0" require one set of operations per cell, variable and time level and must therefore be proportional to $N_{Cells}N_{Var}N_t/N_{Proc}$ as well. Note that we have grouped all operations including addition and assignment in the pseudo code above into one term in the performance model for simplicity.

The complexity of updating the time spectral derivative can finally be seen to be proportional to $N_{Cells}N_{Var}N_t^2/N_{Proc}$. This gives that the time per iteration can be expressed as follows

$$
\begin{aligned}
f(N_{Cells}, N_{Var}, N_t, N_{Proc}) = 3\frac{N_{Cells}N_{Var}N_t}{N_{Proc}}3C_{Flux} + \frac{N_{Cells}N_{Var}N_t}{N_{Proc}}C_{Copy} \\
+ 3\frac{N_{Cells}N_{Var}N_t}{N_{Proc}}N_tC_{Spec} + 3N_{Cells}^{2/3}N_{Var}N_t2C_{MPI} \qquad (16)
\end{aligned}
$$

It should be noted that a 3 was added before the flux, spectral derivative and MPI terms since each of these operations are done once per Runge Kutta stage. The addition and assignment operations are however not done for every Runge Kutta stage so the number of times these operations are done are absorbed into $C_{Copy}$. To make this expression easier to interpret, introduce the variable $N_{Dof} = N_{Cells}N_{Var}N_t/N_{Proc}$ to represent the number of degrees of freedom each process handles. In addition, we define $C_{G3D} = 3C_{Flux} + C_{Copy}/3$ which represents the average cost of performing the flux and addition/assignment operations of one Runge Kutta stage. A different way to see it is that one Runge Kutta stage in the original solver takes $C_{G3D}N_{Cells}N_{Var}/N_{Proc}$ seconds if MPI Communication is neglected. With these definitions the performance model now reads

$$
f(N_{Cells}, N_{Var}, N_t, N_{Proc}) = 3\left[C_{G3D}N_{Dof} + C_{Spec}N_{Dof}N_t + C_{MPI}2N_{Cells}^{2/3}N_{Var}N_t\right] \qquad (17)
$$

## 3 Profiling with Cachegrind

The first profiling of the code was done with the Cachegrind [1.] tool on a Linux Workstation equipped with an Intel Core i7-4770K processor. Cachegrind is a part of the Valgrind package and can be used to get line-by-line information on the number of cache misses as well as the number of intruction reads (Ir). The tool breaks down memory access information into L1 cache misses, including both instruction (I1) and data (D1) cache misses, and lower level data cache misses (DL), representing all cache misses in L2 cache and above. In addition to this, it differentiates between missed read (mr) and missed write (mr) operations, giving four possible cache data misses: D1mr, DLmr, D1mw, DLmw. According to the Cachegrind manual, the most important sources of bad performance are large amount of instruction reads (Ir) and lower level cache misses (DLmr and DLmw). To run the G3D::Flow solver with Cachegrind, both PETSc and the G3D::Flow source code were compiled with the ”-g” flag in addition to the normal compiler flags used.

### 3.1 Test Case

A small test case was designed to enable efficient profiling of the code on the workstation computer. The computational domain consist of one block with $100 \cdot 100 \cdot 10 = 100,000$ cells and a harmonic wave was specified at all boundaries to force the solution inside the domain into a harmonic oscillation. Three harmonics were retained in the Fourier series expansion, giving a Harmonic Balance simulation with $N_t = 7$ time levels. For all simulations performed the number of processes were furthermore set to $N_{Proc} = 2$, giving a total of $50,000$ cells per process. Earlier experience with G3D::Flow has shown that this is a lower bound on the number of cells per processor needed to retain acceptable MPI overhead, altough MPI communication is not the prime focus of the Cachegrind profiling. The computational domain and contours of the $x$ component of velocity are shown in Figure 1.

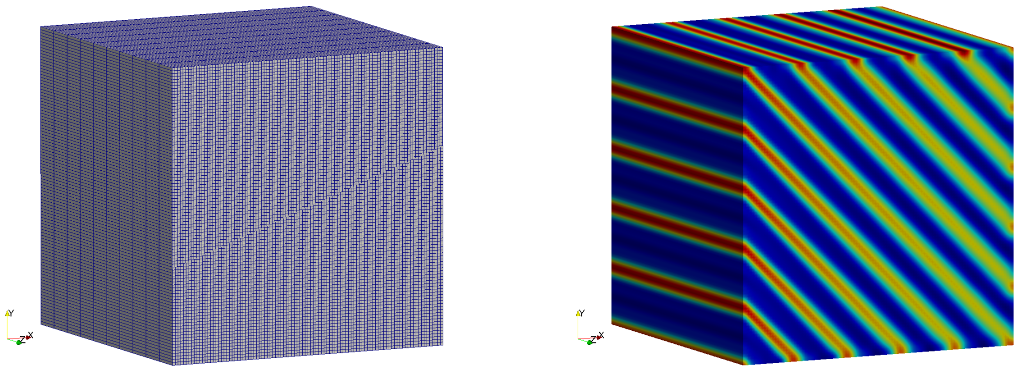

(a) Computational Domain. (b) Axial Velocity ($u$).

Figure 1: Test Case for Cachegrind Profiling.

## 3.2 Results for Original Time Spectral Derivative

As presented in section 2.2 the main change to the code was to repeat the routines of the original G3D::Flow solver $N_t$ times and add the time spectral derivative calculation. This is reflected in the performance model (Eq. (17)), which says that the cost of the Harmonic Balance solver is $N_t$ times the original solver plus the cost of calculating the time spectral derivative. The first objective of the profiling is therefore to find out wheter the new routine "ExplicitSpectralDerivative()" adds any considerable computational cost relative to the rest of the calculation. If not, the focus will be shifted towards the original G3D::Flow routines. A breakdown of the total number of instruction reads and cache misses taking place inside the "ExplicitSpectralDerivative()" routine as percentages of the total number of instruction reads and cache misses are presented in Table 1.

| Ir | D1mr | D1mw | DLmr | DLmw |
|---|---|---|---|---|
| 19.8% | 5% | 0% | 14% | 0% |

Table 1: Instruction reads and cache misses for original time spectral derivative implementaiton.

It is clear from this table that the time spectral derivative calculation both consumes a lot of instructions and causes a relatively large amount of L1 and LL cache read misses. It was found by closer inspection that the majority of all instruction reads and misses took place in the innermost loop, where the residual is updated. These numbers are also expected to grow if the number of time levels are increased. To see this, we assume negligible MPI communication and express the ratio of time spent in the time spectral derivative loop to the total time of execution using Eq. (17)

$$\frac{t_{Spec}}{t_{Tot}} = \frac{N_t}{C_{G3D}/C_{Spec} + N_t} \tag{18}$$

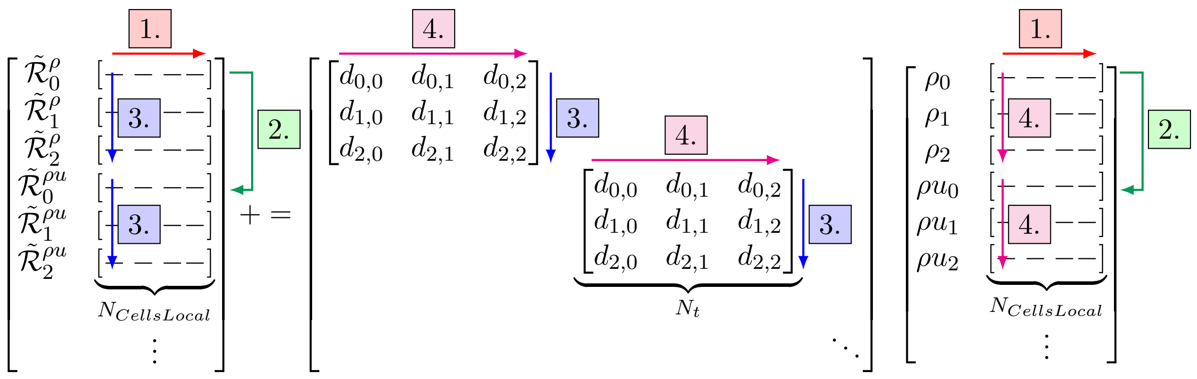
If we assume that the execution time is proportional to the number of instruction reads the above relation becomes equal to Ir in Table 1. This shows that the percentage of instruction reads necessary to compute $DQ^*$ will dominate more and more the larger $N_t$ becomes. It is also clear that increasing the ratio $C_{G3D}/C_{Spec}$ helps alleviating this effect, i.e. reducing the cost of the time spectral derivative calculation ($C_{Spec}$). To achieve this a new block structured matrix multiplication was implemented to compute $D\mathcal{Q}^*$. It turned out to both improve cache locality and reduce number of instruction fetches, as will be described next.

### 3.3 Blocked Time Spectral Derivative

We start by analyzing the original implementation of "ExplicitSpectralDerivative()" and try to identify its shortcomings. The implementation performs a matrix-matrix multiplication between the matrix $D$ and a matrix containing $\mathcal{Q}_i^*$ structured into $N_{CellsLocal}$ columns. Each column is furthermore scaled with $\Delta\tau_i$ and added to a matrix containing the total flow residual $\tilde{\mathcal{R}}_i^*$ orgainzed into $N_{CellsLocal}$ columns. If we for brevity assume that the time step is constant the operation can be summarized as follows

$$[\tilde{\mathcal{R}}_1^* \; \tilde{\mathcal{R}}_2^* \; \ldots \; \tilde{\mathcal{R}}_{N_{Cells}}^*] += \; \Delta\tau D[\mathcal{Q}_1^* \; \mathcal{Q}_2^* \; \ldots \; \mathcal{Q}_{N_{Cells}}^*] \tag{19}$$

The matrix multiplication is done by four nested loops in "ExplicitSpectralDerivative()". In Figure 2 the order of the loops are schematically depicted by numbered arrows, where 1 and 4 denote the outermost and innermost loop respectively. For simplicity the number of time levels are kept to $N_t = 3$.



1.
$\tilde{\mathcal{R}}_0^\rho$
$\tilde{\mathcal{R}}_1^\rho$
$\tilde{\mathcal{R}}_2^\rho$
$\tilde{\mathcal{R}}_0^{\rho u}$
$\tilde{\mathcal{R}}_1^{\rho u}$
$\tilde{\mathcal{R}}_2^{\rho u}$
3.
3.
$N_{CellsLocal}$
2.
+=
4.
$d_{0,0}$ $d_{0,1}$ $d_{0,2}$
$d_{1,0}$ $d_{1,1}$ $d_{1,2}$
$d_{2,0}$ $d_{2,1}$ $d_{2,2}$
3.
4.
3.
$N_t$
1.
$\rho_0$
$\rho_1$
$\rho_2$
$\rho u_0$
$\rho u_1$
$\rho u_2$
4.
4.
$N_{CellsLocal}$
2.

Figure 2: Flow Chart for Time Spectral Derivative Calculation.

The first thing that should be noted from the flowchart is that loop 1 goes through all cells in the mesh. This does in turn mean that loops $2-4$ always will access data from different arrays in memory. When the value in cell $i$ is brought from main memory the cacheline will most likely also contain values for cell $i+1, i+2, \ldots$. The problem with the current implementation is that the value in cell $i+1$ is not used until $\sim N_{Var} \cdot N_t^2$ loop steps later. At this point it may have been replaced in the cache, which then would generate a costly cache miss. This could explain the rather high percentage of L2 cache read misses reported in Table 1. This cache locality issue holds for both the matrix containing the residual and flow field values. It is however not a problem for the time spectral derivative matrix $D$ itself. This is because all $N_{Var}$ blocks inside it are equal, and it thus suffices to store one of them of size $N_t^2$. It is therefore reasonable to believe that the time spectral derivative matrix will not be replaced in the cache at any point of the computation. The cache locality can be improved by implementing a block structured matrix multiplication instead. In this approach the outermost loop instead takes steps of size NCellsBlock >> 1 and a new innermost loop is added that loops serially through all cells in the block. Operations that access objects through pointers (a->b) and dereference data in arrays (a[i]) also consume instruction reads. Minimizing the use of these operations inside the innermost loops is therefore believed to be a key to improve performance. Local pointers that point directly to the data arrays were therefore allocated in "ExplicitSpectralDerivative()" and used instead of the placeholders "Res" and "Cons" to access data. The block structured implementation is presented in pseudo code below

```
void ExplicitSpectralDerivative()
{

  PetscScalar *Residual;
  PetscSCalar *Variable;

  // Loop over all cell blocks of size NCellsBlock
  for(int icb=0; icb<NCellsLocal; icb = icb + NCellsBlock){
    // Loop through all variables (blocks of D)
    for(int ivar=0; ivar<NVar; ivar++){
      // Loop through all residuals
      for(int m=0; m<Nt; m++){
        // Set local pointer
        Residual = Res[m]->Var[ivar];
        // Loop through all time levels
        for(int l=0; l<Nt; l++){
          // Set local pointer
          Variable = Cons[l]->Var[ivar];
          // Loop through all cells in current cell block
          for(int ic=icb; ic<min(NCellsLocal, icb+NCellsBlock); ic++){
            Residual[ic] += dtime[ic]*d[m,l]*Variable[ic];
          }
        }
      }
    }
  }
  return;
}
```

The new loop order can be seen to loop serially through both the "Residual", "dtime" and "Variable" arrays in the innermost loop to improve cache locality. The blocked loop also reduces the amount of pointer and array access operations needed when used in combination with the local variables "Residual" and "Variable". It can for example be seen that the number of times the operation "Cons[l]->Var[ivar]" is done has been reduced by a factor "NCellsBlock" compared to the old code. It must however be recognized that the compiler sometimes seem to realize that a memory location is reused inside a loop and then only computes it once. I.e an operation of type a->b[i]->d appears to be pre-computed before a loop if the loop index is different from i. Another good feature is that the innermost loop now is suitable for vectorization. We will however get back to this later when the code is profiled with Allinea Map. A flowchart of how the new implementation performs the matrix multiplication is presented in Figure 3.

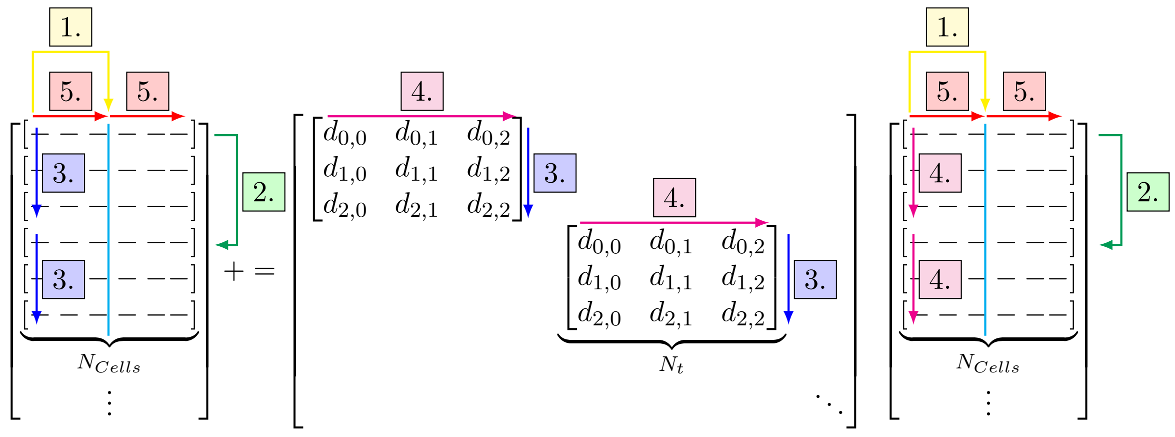

1.
5.
5.
3.
3.
2.
+ =
4.
$d_{0,0}$ $d_{0,1}$ $d_{0,2}$
$d_{1,0}$ $d_{1,1}$ $d_{1,2}$
$d_{2,0}$ $d_{2,1}$ $d_{2,2}$
$N_{Cells}$
$N_t$
4.

Figure 3: Flow Chart for Blocked Time Spectral Derivative Calculation.

Investigation into the performance of the new implementation, including finding the optimal value of "NCellsBlock" is presented in the next section.

### 3.4 Results for Blocked Time Spectral Derivative

In the previous section it was predicted that the number of cache misses and instruction reads should decrease with the new implementation. In fact, increasing NCellsBlock should allways reduce the number of instruction reads by a corresponding factor. From a cache point of view the size of NCellsBlock was however restricted to ensure that all data handled in loops 2-5 would fit the L2 cache for realistic values of $N_t$. Inside the first loop, the code updates the residual for all cells in the current cell block. To do this it uses the corresponding flow field values in the cell block. The total number of doubles that must be stored are therefore $2N_{Var}N_t\cdot$ NCellsBlock. For the applications intended for G3D::Flow it is thought that $N_t$ is in the order of 17, implying that we need to store 170$\cdot$ NCellsBlock doubles at most. In addition to this the time step in all cells must be stored, which becomes NCellsBlock doubles. The time spectral derivative matrix furthermore requires $N_t^2 = 289$ doubles. One double consumes 8 bytes and the L2 cache of a modern CPU is around 256kB. This implies that the largest value of NCellsBlock is restricted by

$$256 \cdot 1024 \geq 8 \cdot (171 \cdot \text{NCellsBlock} + 289). \tag{20}$$

This gives NCellsBlock = 190. To enable a few more features to be implemented into the loop which requires more data (such as damping and variable frequency) we decided to go for a conservative value of NCellsBlock = 64, which represents a third of the "optimal" value derived above. To verify that a large value of NCellsBlock has a positive impact on performance the value was varied between 8 and 64 and the corresponding Cachegrind output was collected with the same setup that was presented in section 3.1. The results are summarized in Table 2

| NCellsBlock | Ir | D1mr | D1mw | DLmr | DLmw |
|---|---|---|---|---|---|
| - | 19.8% | 5% | 0% | 14% | 0% |
| 8 | 10.63% | 2% | 0% | 4.87% | 0% |
| 16 | 8.2% | 1.55% | 0% | 4.14% | 0% |
| 32 | 6.93% | 1.5% | 0% | 3.77% | 0% |
| 64 | 6.29% | 1.31% | 0% | 3.6% | 0% |

Table 2: Instruction reads and cache misses for blocked time spectral derivative implementation.

These numbers confirm the theory that a blocked implementation can reduce both the number of instruction reads and cache data read misses. The theory presented in Eq. (18) was also verified with NCellsBlock = 64 by investigating the influence of $N_t$ on the performance. The results are presented in Table 3.

| $N_t$ | 7 | 9 | 11 | 13 |
|---|---|---|---|---|
| Ir | 6.29% | 7.91% | 9.49% | 11.02% |
| D1mr | 1.31% | 1.59% | 2.57% | 3.44% |
| DLmr | 3.6% | 3.62% | 3.63% | 3.69% |

Table 3: Instruction reads and cache misses for blocked time spectral derivative implementaiton.

These results verify that the cost of applying the time spectral derivative will become more dominant as the number of time levels increase. It should be stressed that the new block structured matrix multiplication did not change the number of operations needed to compute $DQ^*$, it only made it more efficient. The performance model is thus still valid although the value of $C_{Spectral}$ has decreased. We can also see that the number of instruction reads for $N_t = 13$ is about 11% which still is a considerable amount. Thus it is reasonable to continue with more fine grained profiling using Allinea-Map.

## 4 Profiling with Allinea-Map 6.11

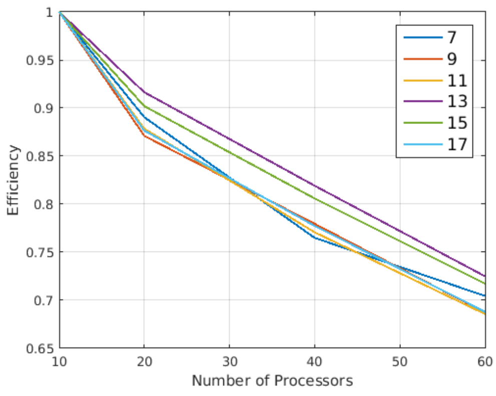
The objective of the Allinea-Map [2.] profiling was to investigate the new implementation on the hardware that G3D::Flow usually is run on, namely the Hebbe and Triolith clusters. The Hebbe compute nodes are equipped with two 10 core Intel Xeon E5-2650 v03 Haswell processors. Triolith have two 8 core Intel Xeon E5-2660 Sandy Bridge processors per node. Two main questions were asked, namely if the code still scales well when run with $50,000$ cells per process and if there are any more performance improvements that can be done to the time spectral derivative calculation.

### 4.1 Test Case

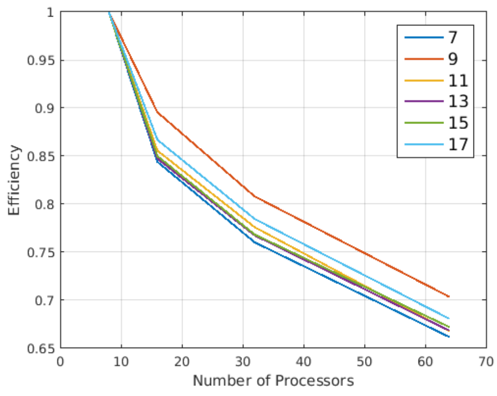
The same basic type of test case was kept for the cluster simulations although the number of cells were increased to $1,000,000$. A matrix of different number of processes ($N_{Proc}$) and time levels ($N_t$) was run on both Hebbe and Triolith to investigate scaling in both dimensions. The number of processes was adjusted on each respective cluster to fit the architecture. The test matrices used on Hebbe and Triolith are presented in Table 4.

| | Hebbe | Triolith |
|---|---|---|
| $N_t$ | 7, 11, 13, 15, 17 | 7, 11, 13, 15, 17 |
| $N_{Proc}$ | 10, 20, 40, 60 | 8, 16, 32, 64 |

Table 4: Matrices used for investigationg scaling on Hebbe and Triolith.



(a) Efficiency on Hebbe.



(b) Efficiency on Triolith.

Figure 4: Efficiency of the G3D::Flow Harmonic Balance Solver for different values of $N_t$.

## 4.2 Results for Blocked Time Spectral Derivative

### 4.2.1 Scaling

For each combination of $N_{Proc}$ and$N_t$ the code was run for 400 iterations and the time per iteration was then computed based on an average over the last 200 iterations. This was done with a script that reads the output file of G3D::Flow where the time/iteration is written out. In other words, I/O and code setup/allocation is not taken into consideration. It was found necessary to start averaging late since the time per iteration for unknown reasons was very high until about 150 iterations had passed on Hebbe. The same behaviour was not observed on Triolith. For each $N_t$ the computational efficiency normalized by the computational efficiency for the lowest process count ($P0$) was calculated according to

$$\eta = \frac{t_P}{t_1 \cdot P} / \frac{t_{P0}}{t_1 \cdot P0} = \frac{t_P}{t_{P0}} \frac{P0}{P} \tag{21}$$

The computational efficiency is a measure of how efficiently the resources are used. If we would have linear scaling, the speedup we would obtain by going from $P0$ to $P$ processes would be

$$S = \frac{t_{P0}}{t_P} = \frac{P}{P0}. \tag{22}$$

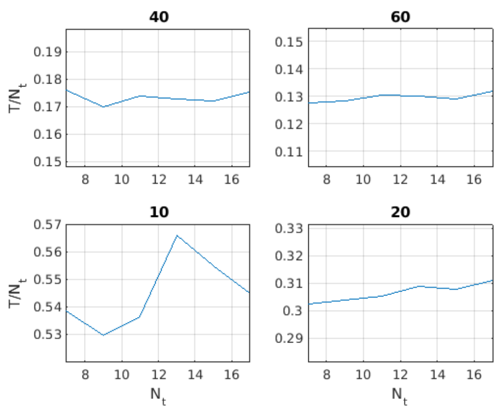
In this case, we are making the most out of the added resources and would also get $\eta = 1$ according to Eq. (21). If linear scaling does not exist, we would instead observe that $\eta$ decreases as $P$ increases. Note that for Hebbe $P0 = 10$ and for Triolith $P0 = 8$. Results from both Hebbe and Triolith are presented in Figure 4.

First it should be pointed out that the efficiency of the code at the lowest number of processes is 1 simply because of the normalization. Secondly it is clear from these figures that no linear scaling is present on either architechture since the efficiency drops as $P$ is increased. On Hebbe the ratio $N_{Cells}/N_{Proc} \leq 100,000$ which implies that the old thumb rule of at least $50,000$ cells per process is not valid anymore. This might in part be due to that considerable work has gone into improving single core performance of G3D::Flow since the old thumb rule was estimated. It is not believed that the new Harmonic Balance solver is responsible for imparing the scaling properties. To see this we use Eq. (17) to express the relative time it takes to perform MPI communication according to

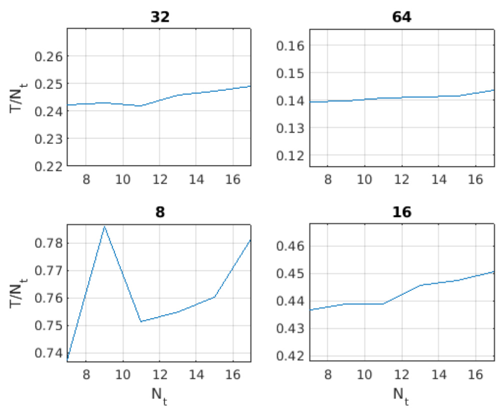
$$\frac{t_{MPI}}{t_{Tot}} = \frac{2C_{MPI}N_{Proc}}{N_{Cells}^{1/3}(C_{G3D} + C_{Spec}N_t) + 2C_{MPI}} \tag{23}$$

Altough it must be recognized that the performance model is rather rough when it comes to estimating MPI communication it demonstrates that if anything, the Harmonic Balance solver should improve the scaling properties since it contributes with a term in the denominator. It is also clear from Figure 4 that the scaling properties did not deteriorate when $N_t$ was increased.

The scaling in the $N_t$ dimension was also investigated by plotting the average time/iteration divided by the number of time levels for all four process counts ($t/N_t$). If we once again refer to the performance model in Eq. (17) we realize that the flux and MPI routines scale linearly with $N_t$ whereas the time spectral derivative scales as $N_t^2$. The time/iteration divided by $N_t$ should thus increase linearly with $N_t$, the larger the slope the more dominant the time spectral derivative calculation is. Results for both clusters are presented in Figure 5.



(a) Scaling on Hebbe.



(b) Scaling on Triolith.

Figure 5: Scaling in $N_t$ of the G3D::Flow Harmonic Balance Solver for different values of $N_{Proc}$.

Altough there are some considerable fluctuations for the lowerst number of processors the graphs confirm the theory presented.

### 4.2.2 Profiling

The last part of the work focused on profiling the code on the Hebbe cluster with Allinea-Map 6.11. No other profiling tool was available on Hebbe, but from our viewpoint this was not a problem since Allinea Map turned out to be very user friendly and offer a great variety of features for investigating performance. First the G3D::Flow source code was compiled with the -g flag in addition to the normal optimization flags used. After this the profiling was done by executing the code with the command "map –profile mpirun ...". Since no operating point in the matrix tested earlier stood out it was decided to profile the code with $N_t = 13$ and $N_{Proc} = 20$. The simulation was once again run for 400 iterations and the output file was post processed with the Allinea-Map GUI.

After getting acquainted with the software it was fairly easy to narrow down on the hotspots in the code and find out exactly how much time different routines took. A good feature of Allinea-Map is that it according to the provider only adds about 5% wall clock time to program execution. This should allow MPI hotspots to be detected without the profiler altering the execution balance significantly. Altough no MPI imbalance was found it was clear that MPI had started to play an important role for this case with $N_{CellsLocal} = 50,000$. A few other places in the code that were not connected to the Harmonic Balance solver were also quickly identified to consume more time than expected. These will however not be reported in this work but will definately be adressed in the near future.

Most importantly for this work is the fact that the "ExplicitSpectralDerivative()" routine consumed almost 6% of the execution time. An interesting feature that Allinea-Map discovered was that the innermost loop was not vectorized by the compiler. It can be argued that vectorizing this loop will not give a significant performance gain, but for an educational purpose (or simply because of the challenge) it was still decided to do it.

## 4.3 Blocked Time Spectral Derivative with Vectorized Loop

According to a guide on Intel's home page [3.] there are a few requirements that should be satisified before a loop can be vectorized by a compiler.

1. First of all the loop should be countable, meaning that there for example should be no conditional break statements inside the loop.
2. Each loop step should perform the same operation, implying that there are no if statements inside of the form "if(i>5) ...".
3. Generally only the innermost loop of a nest can be vectorized.
4. There should be no function calls (except for a few standard math calls for which vectorized functions exist).

In our case all these requirements are satisified which implies that there is something else preventing vectorization. Intel has a list of known issues that prevent vectorization as well [3.]

1. Non contiguous memory access due to that vectorization may be less efficient in these cases
2. Data dependency of different sorts

The first item is not a problem in out case, but the second one might be. There are several different types of data dependencies that can prevent vectorization. Once such issue is that the compiler may not be able to figure out if the memory of two arrays overlap, i.e. that for example a[i] and b[i+1] point to the same memory location. This will prevent vectorization since the compiler does not know if it's safe to update a[i] before b[i+1], which indeed can happen when several loop steps are executed in parallel. If the user knows that all arrays does not overlap it can tell the compiler this by adding "#pragma ivdep;" before the loop. In addition to adding this pragma we also made sure that only one loop index was used (ic) and that all arrays inside the loop were of the same type (PetscScalar). Altough this is the case for the pseudo code presented in this report, in the real program a few more features have been added inside the loop connected to variable frequency and damping. In the original version of the program these were stored in the std::vector datatype and indexed by a different index than the "Residual" and "Variable" arrays. Changing from std::vector to "PetscScalar" arrays, making use of only one loop index and adding the pragma turned out to enable vectorization. To verify that indeed the loop had been vectorized the code was compiled with the flag "-qopt-report=2 -qopt-report-phase=vec". This prompts the Intel compiler to output an optimization report, were indeed it could be seen that the loop had been vectorized. Profiling with Allinea-Map also confirmed that only "vector floating point" operations were executed inside the loop.

## 4.4 Final Results

Allinea-Map was finally used to profile the different implementations of "ExplicitSpectralDerivative()" for a set of $N_t$ and $N_{Proc}$. During every step of the development process the code was tracked with Git, so it was easy to go back to earlier versions of the implementation, compile them and measure performance. Three instances of the code were profiled: the initial implementation, the blocked implementation with NCellsBlock=64 and finally the blocked version with a vectorized inner loop. Results showing the relative time spent inside "ExplicitSpectralDerivative()" are shown in Table 5.

| | $N_t = 7$ | $N_t = 7$ | $N_t = 13$ | $N_t = 13$ |
|---|---|---|---|---|
| | $N_{Proc} = 20$ | $N_{Proc} = 40$ | $N_{Proc} = 20$ | $N_{Proc} = 40$ |
| No Optimization | 14% | 12.2% | 21.1% | 19.2% |
| Blocked | 3.8% | 3.1% | 5.2% | 4.9% |
| Vectorized | 2.1% | 1.7% | 2.8% | 2.1% |

Table 5: Evolution of performance for time spectral derivative calculation.

## 5 Conclusions

In this work a Harmonic Balance solver has been implemented into an existing CFD solver. The aim of the implementation is to efficiently study time periodic problems by solving a set of steady state equations instead of finding the evolution in time using the already present time accurate Runge-Kutta solver. The most important outcome of this work is that the performance of the implementation could be drastically improved by fairly small means. The three main contributors to better performance were improved cache locality, reduced number of instruction reads and a vectorized loop.

An interesting aspect of this work is that todays tools such as Cachegrind and Allinea-Map make the job of finding hot spots in the code easy. It only takes a compiler flag and a few commands before the user can get in-depth information on program execution, including single core and communication times. Although finding the hot-spots is easy, acting upon this information is far more difficult. It for example has to be decided if it is reasonable that a routine consumes a lot of time and if not, what to do about it. In our example the remedy was rather clear but in other situations a deeper analysis into how the computer works may be needed in order to come up with an efficient solution. Nevertheless it is in the authors opinion always a good idea to profile the code, given that it's both quick and easy. Even if no remedies can be found at once, learning about the code instead of guessing what is efficient or not is definately useful knowledge.

## 6 References

1. Cachegrind, Manual:
   http://valgrind.org/docs/manual/valgrind_manual.pdf
2. Allinea-Map, Manual:
   http://content.allinea.com/downloads/userguide-forge.pdf
3. Intel, A Guide to Vectorizing with Intel C++ Compilers:
   https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf