2060042 T/AW084/21 D3.1: Report on DG performance in Nektar++

Edward Laughton, University of Exeter
David Moxey, King's College London
Chris Cantwell & Spencer Sherwin, Imperial College London

7th April 2022

Contents

1 Introduction

1.1 Executive summary

This report outlines the current single-node performance of discontinuous Galerkin (DG) operators within Nektar++ on x86-64 CPU hardware platforms, with the aim of identifying present bottlenecks in the development of higher-dimensional DG kernels. In the remainder of this introduction, we outline the mathematical formulation for the DG formulation. In particular, we aim to profile a realistic test case of the compressible Navier-Stokes equations in order to indicate areas for performance improvements in real-world simulation cases. Section 2 outlines the profiling methodology, and results are indicated in Section 3. Both communication and compute profiling is considered, although these tests are performed in the setting of a single node since the focus of this task in single-node performance. Finally, Section 4 offers brief conclusions and areas for development.

1.2 Discontinuous formulation

This work was undertaken on the compressible Navier-Stokes equations, discretised using the DG method. This section gives a brief overview of the method for the purposes of indicating particular hotspots within the code in the following section; a thorough overview can be found in several other works, e.g. [1, 2, 3]. Considering a three-dimensional domain $\Omega \subset \mathbb{R}^3$ comprised of non-overlapping elements Ω^e such that $\Omega = \bigcup_e \Omega^e$ and given a general hyperbolic conservation law for conserved variables $\boldsymbol{u}$ taking the form

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{u}) = 0, \tag{1}$$

where the flux function $\boldsymbol{F}(\boldsymbol{u}) = (\boldsymbol{f}(\boldsymbol{u}), \boldsymbol{g}(\boldsymbol{u}), \boldsymbol{h}(\boldsymbol{u}))$, we construct the weak form on a single element via multiplication by a test function $\boldsymbol{v}$ and integrating by parts to obtain

$$\left(\boldsymbol{v}, \frac{\partial \boldsymbol{u}}{\partial t}\right)_{\Omega^e} + \left\langle \boldsymbol{v}\boldsymbol{n}, \tilde{\boldsymbol{f}}(\boldsymbol{u}^+, \boldsymbol{u}^-)\right\rangle_{\partial\Omega^e} - (\nabla \boldsymbol{v}, \boldsymbol{F}(\boldsymbol{u}))_{\Omega^e} = 0, \tag{2}$$

where $(u, v)_{\Omega^e} = \int_{\Omega^e} uv \, d\boldsymbol{x}$ and $\langle u, v\rangle_{\partial\Omega^e} = \int_{\partial\Omega^e} uv \, ds$ denote inner products on the volume and surface, respectively. $\tilde{\boldsymbol{f}}$ defines a numerically-calculated flux term which depends on the element-exterior and interior variables $\boldsymbol{u}^+$ and $\boldsymbol{u}^-$, respectively. Using the notation of Karniadakis & Sherwin [3], this can also be written in matrix form for one component of $\boldsymbol{u}$ as

$$\begin{aligned}\frac{\mathrm{d}\hat{\boldsymbol{u}}^e}{\mathrm{d}t} = [\boldsymbol{M}^e]^{-1} \Big[\left(\boldsymbol{D}^e_{x_1}\boldsymbol{B}^e\right)^\intercal \boldsymbol{W}^e \boldsymbol{\Lambda}^e \left(f\left(u\right)\right) + \left(\boldsymbol{D}^e_{x_2}\boldsymbol{B}^e\right)^\intercal \boldsymbol{W}^e \boldsymbol{\Lambda}^e \left(g\left(u\right)\right) \\ + \left(\boldsymbol{D}^e_{x_3}\boldsymbol{B}^e\right)^\intercal \boldsymbol{W}^e \boldsymbol{\Lambda}^e \left(h\left(u\right)\right)\Big] - [\boldsymbol{M}^e]^{-1} \boldsymbol{b}^e\end{aligned} \tag{3}$$

where $\boldsymbol{M}^e$ denotes an elemental mass matrix, the collection of matrix multiplications in the square bracketed term denotes the volumetric inner product with respect to the derivative of the basis functions, and $\boldsymbol{b}$ is a vector corresponding to the surface integral such that

$$\boldsymbol{b}^e\left[n(pqr)\right] = \int_{\partial\Omega^e} \phi_{pqr} \tilde{\boldsymbol{f}} \cdot \boldsymbol{n}^e \mathrm{d}s \tag{4}$$

of which $n(pqr)$ is the mapping from local tensor-product basis indices p, q and r to a consecutive numbered list. We represent $\boldsymbol{u}$ using an expansion of high-order polynomials, so that

$$\boldsymbol{u}^\delta = \sum_n \hat{\boldsymbol{u}}_n \phi_n \left([\boldsymbol{\chi}^e]^{-1}(\boldsymbol{x})\right). \tag{5}$$

In this expression, we note that the approximation is defined with the use of a standard (reference) element Ω_{st}, with ϕ_n denoting an appropriate set of basis functions. An isoparametric mapping $\boldsymbol{\chi}^e : \Omega_{\mathrm{st}} \to \Omega^e$ defines a possibly curvilinear element Ω^e, so that $\boldsymbol{x} = \boldsymbol{\chi}^e(\boldsymbol{\xi})$ for $\boldsymbol{\xi} \in \Omega_{\mathrm{st}}$. We additionally equip the standard element with a distribution of quadrature points $\boldsymbol{\xi}_q$ and weights w_q, so that upon selecting test functions $v = \phi_n$ we then evaluate the terms in eq. (2) as finite summations, i.e.

$$(\nabla \phi_n, \boldsymbol{F}(\boldsymbol{u}))_{\Omega^e} \approx \sum_q \nabla \boldsymbol{\chi}^e(\boldsymbol{\xi}_q)^{-\intercal} \nabla \phi_n(\boldsymbol{\xi}_q) \cdot \boldsymbol{F}(\boldsymbol{u}(\boldsymbol{x}_q)) \det\left(\boldsymbol{\chi}^e(\boldsymbol{\xi}_q)\right) w_q \tag{6}$$

Following from this general formulation the Navier-Stokes equations in conservative form expressed similarly to eq. (1) can be written as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{u}) = \nabla \cdot \boldsymbol{F}_v(\boldsymbol{u}, \nabla \boldsymbol{u}), \tag{7}$$

where $\boldsymbol{u} = [\rho, \rho u, \rho v, \rho w, E]$ is the vector of conserved variables in terms of density ρ, velocity $\boldsymbol{v} = (u_1, u_2, u_3) = (u, v, w)$ and E is the specific total energy. In three dimensions we have that

$$\boldsymbol{F}(\boldsymbol{u}) = \begin{bmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ u(E+p) & v(E+p) & w(E+p) \end{bmatrix}, \tag{8}$$

To close the system we need to specify an equation of state; in this case we use the ideal gas law $p = \rho RT$ where T is the temperature and R is the gas constant. The tensor of viscous forces $\boldsymbol{F}_v(\boldsymbol{u}, \nabla \boldsymbol{u})$ is defined as

$$\boldsymbol{F}_v(\boldsymbol{u}, \nabla \boldsymbol{u}) = \begin{bmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \tau_{yy} & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \tau_{zz} \\ A & B & C \end{bmatrix}, \tag{9}$$

with

$$\begin{aligned} A &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\partial_x T, \\ B &= u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\partial_y T, \\ C &= u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\partial_z T, \end{aligned}$$

where in tensor notation the stress tensor $\tau_{x_i x_j} = 2\mu(\partial_{x_i} u_i + \partial_{x_i} u_j - \frac{1}{3}\partial_{x_k} u_k \delta_{ij})$, μ is the dynamic viscosity calculated using Sutherland's law, k is the thermal conductivity and δ_{ij} is the Kronecker delta. We note that the inclusion of the viscous term $\boldsymbol{F}_v(\boldsymbol{u}, \nabla \boldsymbol{u})$ requires additional treatment, in particular a careful selection of flux terms in order to preserve spatial accuracy. In the simulations below, we adopt the local discontinuous Galerkin (LDG) approach, wherein an auxiliary variable $\boldsymbol{q} = \nabla \boldsymbol{u}$ is introduced and discretised alongside equation (7). With a careful choice of alternating fluxes (so that $\tilde{\boldsymbol{q}} = \boldsymbol{q}^+$ and $\tilde{\boldsymbol{u}} = \boldsymbol{u}^-$, or vice versa), high-order accuracy can be preserved [4].

1.3 Case setup

A three-dimensional Taylor-Green vortex was chosen as the most suitable case to profile. Although it is certainly more representative of realistic fluid dynamics simulations, it is also the highest spatial dimension presently supported by Nektar++. This is also important from the perspective of parallelisation. Assuming that the basis functions admit a boundary-interior decomposition, the volume of data to be communicated between processors is proportional to the number of modes which lie on the boundary facets of the element, which are called the 'trace' or 'skeleton' of the mesh. In moving up a dimension from 2D to 3D simulations, the volume of data to be transferred is now a power of 2 larger, i.e. going from a 1D segment of order $\mathcal{O}(p)$ to a 2D quadrilateral or triangle of order $\mathcal{O}(p^2)$, significantly increasing the complexity of mesh partitioning and the volume of communicated data.

Although the precise choice of test case is typically relatively unimportant in the context of performance profiling, we have elected to consider a benchmark test case for fluid dynamics solvers: the Taylor-Green vortex at Re = 1600. In this case, starting vortices are defined in a periodic box $\Omega = [-L\pi, L\pi]^3$, given a reference length L, which break down into turbulent eddies before decaying due to viscous effects. The initial conditions are given in primitive variables $(\boldsymbol{v}, p)$ as

$$\begin{aligned} u &= V_\infty \sin(x/L)\cos(y/L)\cos(z/L), \\ v &= -V_\infty \cos(x/L)\sin(y/L)\cos(z/L), \\ w &= 0, \\ p &= \rho_\infty V_\infty^2 \left[\frac{1}{\gamma \mathrm{Ma}_\infty^2} + \frac{1}{16}\left(\cos(2x/L)+\cos(2y/L)\right)\cdot\left(\cos(2z/L)+2\right) \right], \end{aligned}$$

with the Reynolds number $\mathrm{Re} = \rho_\infty U_\infty L/\mu$ and the Prandtl number $\mathrm{Pr} = 0.71$. Although the Taylor-Green vortex problem is traditionally examined in the setting of an incompressible flow, we approach this limit by considering flows with low compressibility effects so that the Mach number $\mathrm{Ma}_\infty = 0.1$. We select an explicit fourth-order Runga-Kutta time integration scheme, with the time-step fixed at 2×10^{-5} and the case is run for a quarter of the time interval of $t_c \in [0, 20]$ where $t_c = tU_\infty/L$ is the convective timescale, in this case corresponding to 7347 time-steps. We also make use of the exact Riemann solver of Toro [5] to evaluate the numerical flux term $\tilde{\boldsymbol{f}}$. An $8 \times 8 \times 8$ hexahedral mesh is used, shown in figure 1, with the modified basis functions of Karniadakis & Sherwin (eq. 3.1.1 in [3]) with order 6.

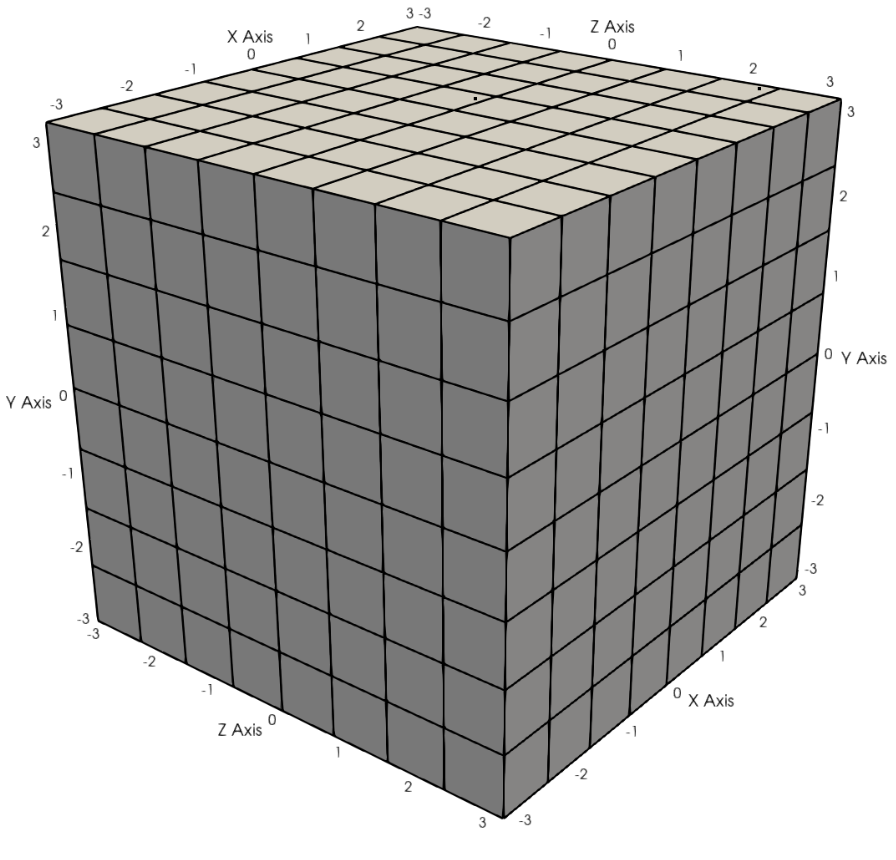


Figure 1: $8 \times 8 \times 8$ hexahedral 3D mesh for the Taylor-Green vortex case.

2 Profiling software and hardware setup

The profiling of the compressible flow solver in Nektar++ was performed on the latest master branch compiled in release mode with debug symbols attached, i.e. `RelwithDebInfo`. This still applies `O2` level optimisation but allows for more descriptive data to be gathered from the sampling done by the profiler. It is also worth noting that the `O2` level optimisation will inline functions and so the call trees may be missing some of the deeper functions, however this will not change the overall picture as the major patterns will still be evident.

Oracle Developer Studio was used as the profiling tool, because it has the capability to profile processes running using MPI, allowing for a more in-depth look at the MPI communication patterns. It is worth emphasising that the profiler introduces some substantial overhead to the communication when tracing the MPI calls for each individual process, and therefore for the main profiling of 'hotter' functions during the simulation, this case is also run without the MPI tracing enabled to better look at the raw performance. Nektar++ makes use of tuning to pick the fastest MPI method dynamically for the given system it is being run on: the four methods implemented are `AllToAll`, `AllToAllV`, `PairwiseSendRecv`, and `NeighborAllToAllV`. Each of these implements a slightly different communication strategy. Of these methods, the `PairwiseSendRecv` and `NeighborAllToAllV` are usually the fastest, unless with exactly symmetric partitioning at core counts ≤ 8. Unfortunately the creation of the communicator with the distributed graph topology attached, which is needed for the collective MPI-3.0 `NeighborAllToAllV` method, breaks the MPI tracing as it is still attached to the old communicator, so the `PairwiseSendRecv` was forced for MPI tracing case. Oracle Developer Studio was also already installed on a suitable machine which can be challenging on a shared academic system due to the level of permissions required for the necessary sampling potentially introducing security vulnerabilities.

The profiling was undertaken on a dual-socket Broadwell micro-architecture Intel Xeon E5-2697 v4 system with 18 physical cores per socket, for a total of 36 physical cores across two NUMA domains, and 192GB RAM. During profiling, only 18 cores were used; bound to a single socket by using the `mpirun -np 18 --bind-to socket` command to prevent processes dynamically switching cores and adding unnecessary overhead.

3 Results

3.1 MPI

Figure 2 shows a subsection of the MPI communication timeline with all 18 processes in the vertical direction, with the horizontal axis representing time and black lines indicating inter-process communication using MPI. The MPI functions are coloured while the main Nektar++ functions are labelled "Application". This subsection represents a single RK4 timestep, with the cyan boxes on the far left and right indicating a `MPI_AllReduce` checking for the presence of an abort file. The four RK4 stages are clearly visible as clusters of two communication groups, an initial larger followed by a secondary smaller; both groups consist of a number of pairwise communications between all individual processes that share a mesh partition boundary. In each communication group the variable fields are communicated consecutively, so for the compressible flow solver in 3D we operate on the ρ, ρu, ρv, ρw and the E field in that order.

The first large communication group consists of a call to `DisContField::GetFwdBwdTracePhys` in the `AdvectionLDGNS::NumericalFluxO2` function. As the name suggests, this combination of functions extracts the $\boldsymbol{q}^{+}$ and $\boldsymbol{q}^{-}$ terms, meaning that this is a two-way communication which fills the auxiliary variable q, as needed in the LDG method, trace space from neighbouring elements that are across the mesh partition boundary. It is particularly worth emphasising that using $\boldsymbol{q} = \nabla \boldsymbol{u}$ in the LDG formulation we artifically increase the time-advected variables by d per field variable, where d is the problem dimension. So in this 3D case by including the diffusion component it increases the number of advection equations from one to four for each conserved variable, leading to additional communication overhead and computational cost associated with the additional fields.

The second communication group starts with an `DisContField::ExtractTracePhys` in the `CompressibleFlowSolver::SetBoundaryConditions` routine, which is used to fill the trace on the boundary elements. This communication is not actually needed, since by definition a boundary trace is not connected to any other element (unless it is periodic, but this special case is handled in the `DisContField::GetFwdBwdTracePhys` call subsequently), and so wholly exists on the current process. This unwanted communication occurs because the `ExtractTracePhys` function is generically written to get the `Fwd` trace space in the domain from the physical fields, which means it also fills the spaces across the partitions and thus needs the communication with neighbouring processes. This is then followed by another `GetFwdBwdTracePhys` in `DoOdeRhs` which is a two-way communication to fill the `Bwd` trace space from neighbouring elements `Fwd` trace space that are across the mesh partition boundary. The `Fwd` and `Bwd` are analogous to the element-exterior and interior velocities $\boldsymbol{u}^+$ and $\boldsymbol{u}^-$ in equation (2) and are used to calculate the numerical flux $\tilde{\boldsymbol{f}}$ using a Riemann solver.

The amount of time spent in the MPI processes when compared to the remaining intra-process compute is exaggerated in figure 2 due to the overhead required when sampling the MPI communication and tracing the processes. As noted above, a second profiling case was run with the MPI tracing disabled. It is then possible to see what percentage of time was spent in the `PerformExchange` function, which is responsible for the trace communication between processes in all usages above. This was collected using standard timer routines. Across all processes in this case 7.37% of total CPU time was spent in `PerformExchange`. It is also possible to see the split of time for each process independently as shown in figure 3. This shows that the time spent in `PerformExchange` is not equally distributed among processes, and so there is some load imbalance where some processes will be idle waiting for the MPI communication to complete on others. There a number of reasons this could be the case such as an imbalance in the partitioning, or due to certain cores performing better than others (although efforts were made to run sampling on 'clean' cores). If we take the difference between the shortest and longest CPU time, to mimic a worst case scenario, and multiply by the total CPU time spent, we only get 0.3% inefficiency from the MPI communications load imbalancing, which we consider to be acceptable and would not adversely affect total compute time.

3.2 Compressible flow solver

Again using the case with the MPI tracing disabled we can now observe a performance profile for the main elements of the compressible flow solver. Sorting the functions by exclusive CPU time, i.e. time spent in that function alone and not any functions called by it, 4 out of the top 5 most time spent functions are calls to lower-level BLAS linear algebra routines. The Nektar++ function `DiagonalBlockMatrixMultiply` consists of 44.67% inclusive CPU time, using the BLAS routine `dgemv` shown in a leaf-filtered call tree in figure 6. This is called by the Nektar++ `ExpList::MultiplyByElmtInvMass` function which is used in both the `DoDiffusion` (72.81%) and `DoAdvection` (21.78%) segments of the Navier-Stokes compressible flow solver; additionally it is also used in the main `DoSolve` function as `FwdTrans_IterPerExp` (5.42%) to project functions on the physical space to coefficient space.

As the name suggests, this routine performs a block-diagonal inverse mass matrix multiplication, which is required in equation (3) showing the use in the inner-product of the volume and surface, as well as the surface integral. (However we do highlight that the surface and volume terms are collected *before* the mass matrix multiplication is performed.) This split in cost between the diffusion and advection component fairly accurately represents the theoretical expectations as discussed in the LDG formulation above.

Another significant BLAS call is to the matrix-matrix multiplication kernel `dgemm`, which is 9.34% of inclusive CPU time. These methods are used within matrix-free operations inside local elemental operations, namely `StdHexExp::v_IProductWRTBase` (52.23%), `StdHexExp::v_BwdTrans` (25.32%), and `StdQuadExp::v_IProductWRTBase` (22.44%) as shown in figure 7. Of these methods, the `IProductWRTBase` method is used to calculate the inner product of an input array with respect to the elemental basis functions basis. The `BwdTrans` function translates from coefficient space to physical space (performing the action $\boldsymbol{B}\hat{\boldsymbol{u}}$), and is used in both the `DoDiffusion` (78.85%) and `DoAdvection` (21.15%) routines.

Throughout this, we observe that the diffusion component adds significant computational time, and in this case accounts for 60% of the total CPU time compared to 22% for the advection component; this is visualised in the flame chart in figure 4 and the call tree in figure 5. (The remaining execution time is spent in BLAS calls which are involved in these routines but are recorded outside of these regions due to oddities in profiling; as well as a small percentage for other aspects such as limited I/O and projections to/from coefficient space.) For this reason the compressible Euler flow solver will be significantly faster, since it does not need to calculate the diffusion component. This also matches the theoretical expectations expressed in the LDG formulation regarding the number of additional fields to express the auxiliary variable $\boldsymbol{q} = \nabla\boldsymbol{u}$.

4 Conclusion

Overall, the profiling suggests that there are no standout kernels within the code that show obvious signs of requiring major performance improvements. We make the following observations, however:

- Clearly a key focus for optimisation could be the application of the inverse mass matrix $\boldsymbol{M}^{-1}$. The choice of a modified basis leads, in this case, to a full rank mass matrix, and thus the inverse must be precomputed and stored. For hexahedral elements, one route to investigate is the evalution of the inverse mass matrix via matrix-free methods; if the matrix $\boldsymbol{B}$ is chosen to be square by selecting the same number of quadrature points as modes, then the action of $\boldsymbol{M}^{-1}\boldsymbol{u}$ is given by the evaluation $(\boldsymbol{B}^\top\boldsymbol{W}\boldsymbol{B})^{-1}\boldsymbol{u} = (\boldsymbol{B}^{-1}\boldsymbol{W}^{-1}\boldsymbol{B}^{-\top})\boldsymbol{u}$ where $\boldsymbol{W}$ is the diagonal matrix holding quadrature weights and geometric mappings. This can be evaluated in a matrix-free manner by precomputing the inverse basis matrix (which is common to all elements) and then applying sum-factorisation via the `BwdTrans` kernel for hexahedral elements. The action of $\boldsymbol{W}^{-1}$ is readily computed owing to its diagonal nature. Alternatively, it may be worth investigating change of basis kernels between the modified basis and orthogonal bases, which then also may yield approaches that can be considered for unstructured elements.
- The `SetBoundaryConditions` function could be developed further to reduce communication overheads by creating a specific `ExtractTracePhys` function to only extract boundary condition traces, instead of the current generic function which results in unneeded MPI communication.
- It would also be beneficial to overlap the computation of the volume term with the surface MPI communication. This means that instead of waiting for the surface term communication to finalised immediately, only the surface sends are posted with the volume computation occurring concurrently. Then, once the volume computation is completed the volume sends are also posted followed by both the volume and surface receives. This is

a more efficient setup because it allows further computation to proceed while the communication is in progress, effectively reducing the operative MPI communication overhead with less processes idling.

- Additional benchmarking against other codes could prove useful in terms of identifying othe areas of overhead.

References

[1] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J. E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[2] D. Moxey, C. D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kazemi, K. Lackhove, J. Marcon, G. Mengaldo, D. Serson, M. Turner, H. Xu, J. Peiró, R. M. Kirby, and S. J. Sherwin. *Nektar++*: enhancing the capability and application of high-fidelity spectral/hp element methods. *Computer Physics Communications*, 249:107110, 2020.

[3] George Karniadakis and Spencer Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, Oxford, 2 edition, 2005.

[4] Bernardo Cockburn and Chi-Wang Shu. The local discontinuous galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[5] E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer, Berlin, New York, 3rd edition, 2009.

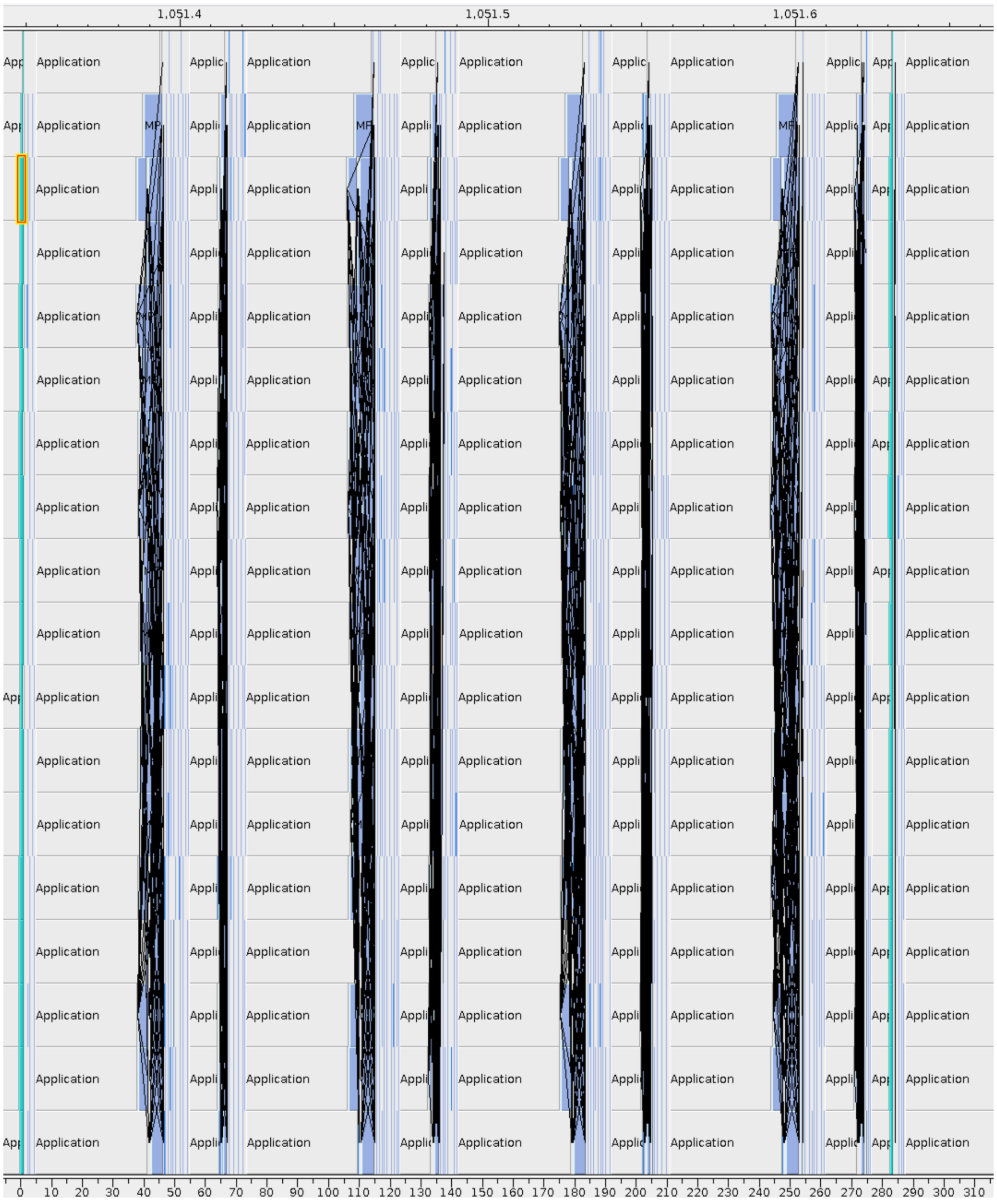
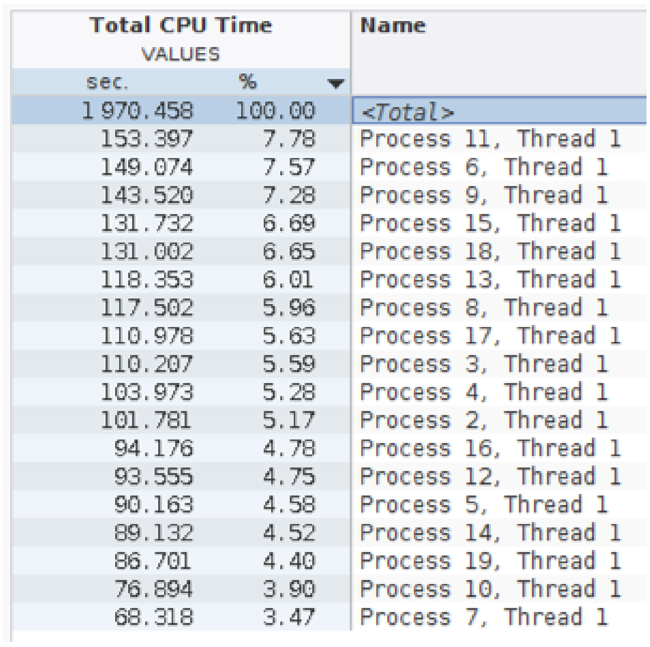


Figure 2: Subsection of MPI communication timeline showing all 18 processes across a single RK4 time-step.



Total CPU Time VALUES sec.	%	Name
1 970.458	100.00	<Total>
153.397	7.78	Process 11, Thread 1
149.074	7.57	Process 6, Thread 1
143.520	7.28	Process 9, Thread 1
131.732	6.69	Process 15, Thread 1
131.002	6.65	Process 18, Thread 1
118.353	6.01	Process 13, Thread 1
117.502	5.96	Process 8, Thread 1
110.978	5.63	Process 17, Thread 1
110.207	5.59	Process 3, Thread 1
103.973	5.28	Process 4, Thread 1
101.781	5.17	Process 2, Thread 1
94.176	4.78	Process 16, Thread 1
93.555	4.75	Process 12, Thread 1
90.163	4.58	Process 5, Thread 1
89.132	4.52	Process 14, Thread 1
86.701	4.40	Process 19, Thread 1
76.894	3.90	Process 10, Thread 1
68.318	3.47	Process 7, Thread 1

Figure 3: Split of time spent in `PerformExchange` across all 18 processes.

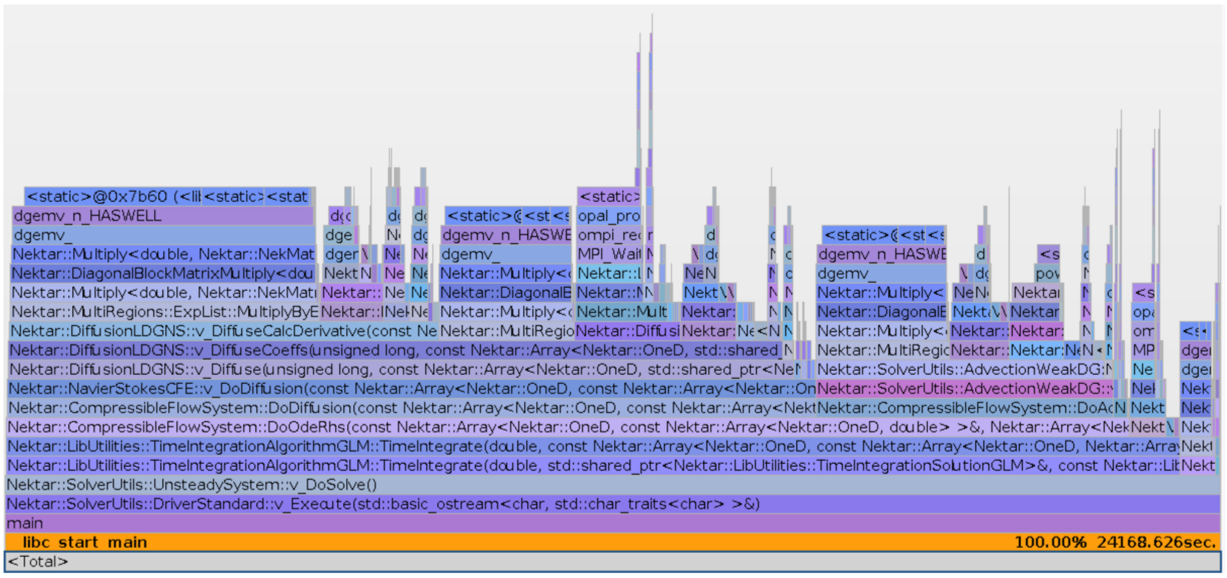
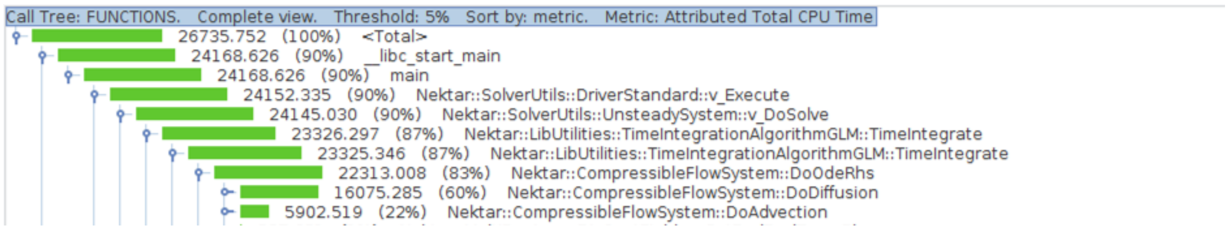
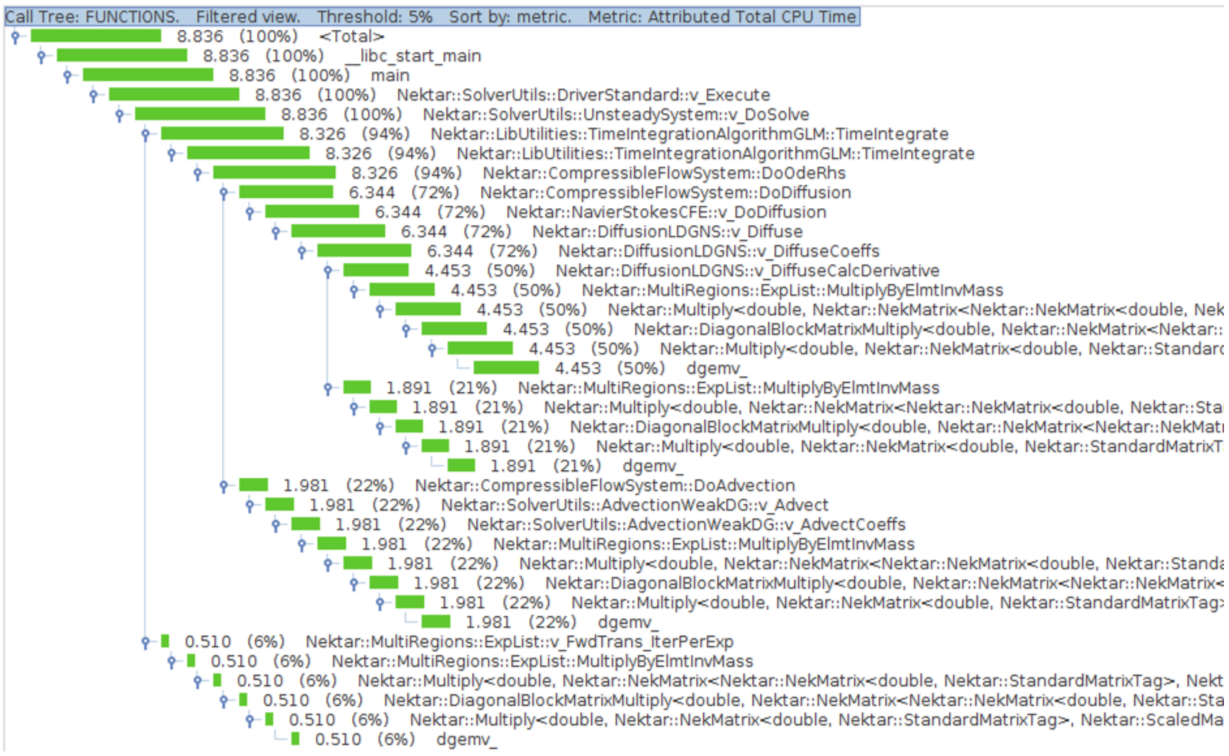


Figure 4: Flame chart showing the breakdown of the stack trace by CPU time spent in individual functions.



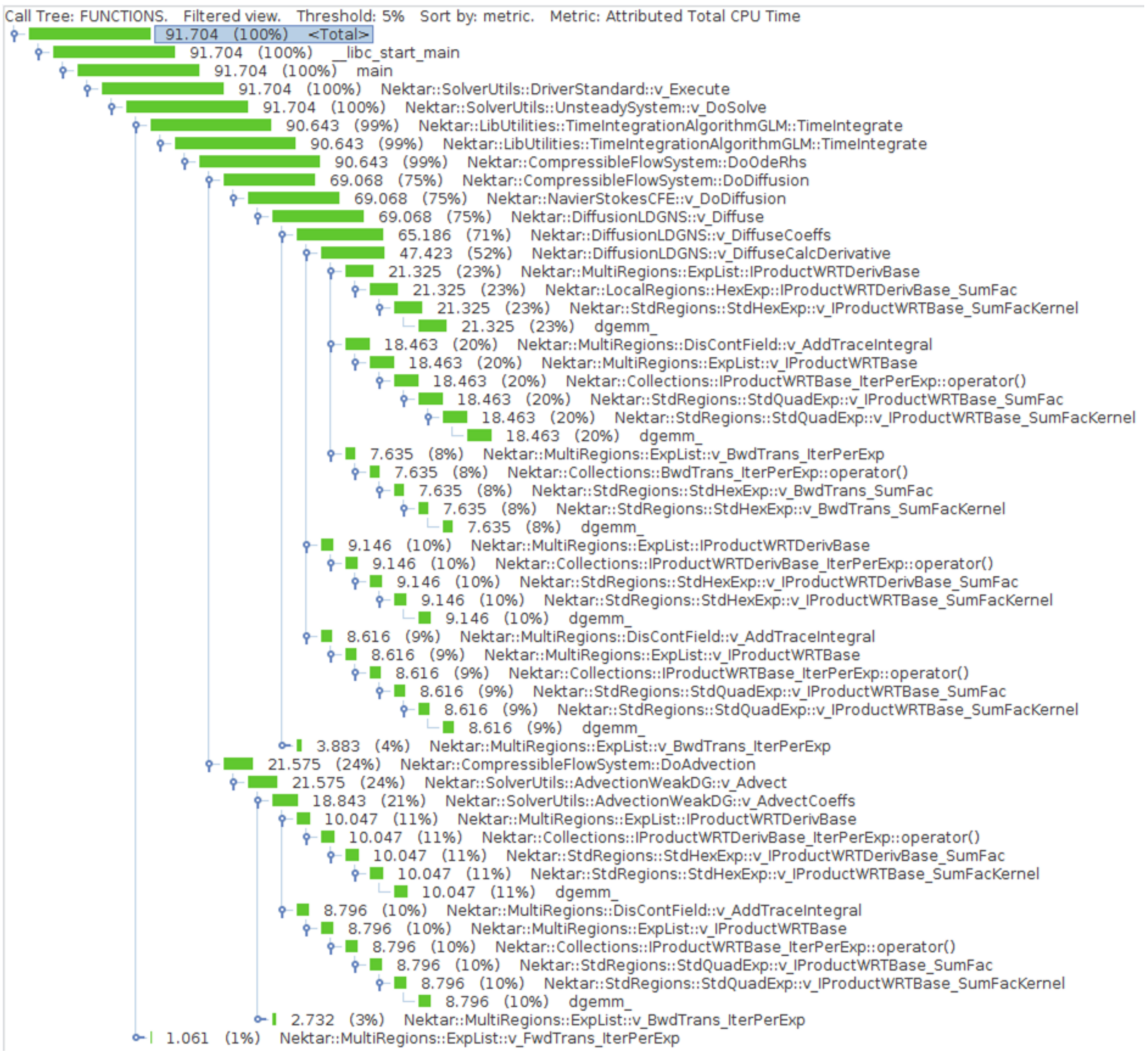
```
Call Tree: FUNCTIONS.  Complete view.  Threshold: 5%  Sort by: metric.  Metric: Attributed Total CPU Time
26735.752  (100%)  <Total>
 24168.626  (90%)  __libc_start_main
  24168.626  (90%)  main
   24152.335  (90%)  Nektar::SolverUtils::DriverStandard::v_Execute
    24145.030  (90%)  Nektar::SolverUtils::UnsteadySystem::v_DoSolve
     23326.297  (87%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
      23325.346  (87%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
       22313.008  (83%)  Nektar::CompressibleFlowSystem::DoOdeRhs
        16075.285  (60%)  Nektar::CompressibleFlowSystem::DoDiffusion
        5902.519  (22%)  Nektar::CompressibleFlowSystem::DoAdvection
```

Figure 5: Call tree showing the CPU time spent on the advection and diffusion components.



```
Call Tree: FUNCTIONS.   Filtered view.   Threshold: 5%   Sort by: metric.   Metric: Attributed Total CPU Time
8.836  (100%)  <Total>
 8.836  (100%)  __libc_start_main
  8.836  (100%)  main
   8.836  (100%)  Nektar::SolverUtils::DriverStandard::v_Execute
    8.836  (100%)  Nektar::SolverUtils::UnsteadySystem::v_DoSolve
     8.326  (94%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
      8.326  (94%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
       8.326  (94%)  Nektar::CompressibleFlowSystem::DoOdeRhs
        6.344  (72%)  Nektar::CompressibleFlowSystem::DoDiffusion
         6.344  (72%)  Nektar::NavierStokesCFE::v_DoDiffusion
          6.344  (72%)  Nektar::DiffusionLDGNS::v_Diffuse
           6.344  (72%)  Nektar::DiffusionLDGNS::v_DiffuseCoeffs
            4.453  (50%)  Nektar::DiffusionLDGNS::v_DiffuseCalcDerivative
             4.453  (50%)  Nektar::MultiRegions::ExpList::MultiplyByElmtInvMass
              4.453  (50%)  Nektar::Multiply<double, Nektar::NekMatrix<Nektar::NekMatrix<double, Nek
               4.453  (50%)  Nektar::DiagonalBlockMatrixMultiply<double, Nektar::NekMatrix<Nektar::
                4.453  (50%)  Nektar::Multiply<double, Nektar::NekMatrix<double, Nektar::Standard
                 4.453  (50%)  dgemv_
            1.891  (21%)  Nektar::MultiRegions::ExpList::MultiplyByElmtInvMass
             1.891  (21%)  Nektar::Multiply<double, Nektar::NekMatrix<Nektar::NekMatrix<double, Nektar::Sta
              1.891  (21%)  Nektar::DiagonalBlockMatrixMultiply<double, Nektar::NekMatrix<Nektar::NekMat
               1.891  (21%)  Nektar::Multiply<double, Nektar::NekMatrix<double, Nektar::StandardMatrixT
                1.891  (21%)  dgemv_
        1.981  (22%)  Nektar::CompressibleFlowSystem::DoAdvection
         1.981  (22%)  Nektar::SolverUtils::AdvectionWeakDG::v_Advect
          1.981  (22%)  Nektar::SolverUtils::AdvectionWeakDG::v_AdvectCoeffs
           1.981  (22%)  Nektar::MultiRegions::ExpList::MultiplyByElmtInvMass
            1.981  (22%)  Nektar::Multiply<double, Nektar::NekMatrix<Nektar::NekMatrix<double, Nektar::Standa
             1.981  (22%)  Nektar::DiagonalBlockMatrixMultiply<double, Nektar::NekMatrix<Nektar::NekMatrix<
              1.981  (22%)  Nektar::Multiply<double, Nektar::NekMatrix<double, Nektar::StandardMatrixTag>
               1.981  (22%)  dgemv_
     0.510  (6%)  Nektar::MultiRegions::ExpList::v_FwdTrans_IterPerExp
      0.510  (6%)  Nektar::MultiRegions::ExpList::MultiplyByElmtInvMass
       0.510  (6%)  Nektar::Multiply<double, Nektar::NekMatrix<Nektar::NekMatrix<double, Nektar::StandardMatrixTag>, Nekt
        0.510  (6%)  Nektar::DiagonalBlockMatrixMultiply<double, Nektar::NekMatrix<Nektar::NekMatrix<double, Nektar::Sta
         0.510  (6%)  Nektar::Multiply<double, Nektar::NekMatrix<double, Nektar::StandardMatrixTag>, Nektar::ScaledMa
          0.510  (6%)  dgemv_
```

Figure 6: Call tree filtered by the dgemv BLAS function.



```
Call Tree: FUNCTIONS.  Filtered view.  Threshold: 5%  Sort by: metric.  Metric: Attributed Total CPU Time
91.704  (100%)  <Total>
  91.704  (100%)  __libc_start_main
    91.704  (100%)  main
      91.704  (100%)  Nektar::SolverUtils::DriverStandard::v_Execute
        91.704  (100%)  Nektar::SolverUtils::UnsteadySystem::v_DoSolve
          90.643  (99%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
            90.643  (99%)  Nektar::LibUtilities::TimeIntegrationAlgorithmGLM::TimeIntegrate
              90.643  (99%)  Nektar::CompressibleFlowSystem::DoOdeRhs
                69.068  (75%)  Nektar::CompressibleFlowSystem::DoDiffusion
                  69.068  (75%)  Nektar::NavierStokesCFE::v_DoDiffusion
                    69.068  (75%)  Nektar::DiffusionLDGNS::v_Diffuse
                      65.186  (71%)  Nektar::DiffusionLDGNS::v_DiffuseCoeffs
                        47.423  (52%)  Nektar::DiffusionLDGNS::v_DiffuseCalcDerivative
                          21.325  (23%)  Nektar::MultiRegions::ExpList::IProductWRTDerivBase
                            21.325  (23%)  Nektar::LocalRegions::HexExp::IProductWRTDerivBase_SumFac
                              21.325  (23%)  Nektar::StdRegions::StdHexExp::v_IProductWRTBase_SumFacKernel
                                21.325  (23%)  dgemm_
                          18.463  (20%)  Nektar::MultiRegions::DisContField::v_AddTraceIntegral
                            18.463  (20%)  Nektar::MultiRegions::ExpList::v_IProductWRTBase
                              18.463  (20%)  Nektar::Collections::IProductWRTBase_IterPerExp::operator()
                                18.463  (20%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFac
                                  18.463  (20%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFacKernel
                                    18.463  (20%)  dgemm_
                          7.635  (8%)  Nektar::MultiRegions::ExpList::v_BwdTrans_IterPerExp
                            7.635  (8%)  Nektar::Collections::BwdTrans_IterPerExp::operator()
                              7.635  (8%)  Nektar::StdRegions::StdHexExp::v_BwdTrans_SumFac
                                7.635  (8%)  Nektar::StdRegions::StdHexExp::v_BwdTrans_SumFacKernel
                                  7.635  (8%)  dgemm_
                        9.146  (10%)  Nektar::MultiRegions::ExpList::IProductWRTDerivBase
                          9.146  (10%)  Nektar::Collections::IProductWRTDerivBase_IterPerExp::operator()
                            9.146  (10%)  Nektar::StdRegions::StdHexExp::v_IProductWRTDerivBase_SumFac
                              9.146  (10%)  Nektar::StdRegions::StdHexExp::v_IProductWRTBase_SumFacKernel
                                9.146  (10%)  dgemm_
                        8.616  (9%)  Nektar::MultiRegions::DisContField::v_AddTraceIntegral
                          8.616  (9%)  Nektar::MultiRegions::ExpList::v_IProductWRTBase
                            8.616  (9%)  Nektar::Collections::IProductWRTBase_IterPerExp::operator()
                              8.616  (9%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFac
                                8.616  (9%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFacKernel
                                  8.616  (9%)  dgemm_
                      3.883  (4%)  Nektar::MultiRegions::ExpList::v_BwdTrans_IterPerExp
                21.575  (24%)  Nektar::CompressibleFlowSystem::DoAdvection
                  21.575  (24%)  Nektar::SolverUtils::AdvectionWeakDG::v_Advect
                    18.843  (21%)  Nektar::SolverUtils::AdvectionWeakDG::v_AdvectCoeffs
                      10.047  (11%)  Nektar::MultiRegions::ExpList::IProductWRTDerivBase
                        10.047  (11%)  Nektar::Collections::IProductWRTDerivBase_IterPerExp::operator()
                          10.047  (11%)  Nektar::StdRegions::StdHexExp::v_IProductWRTDerivBase_SumFac
                            10.047  (11%)  Nektar::StdRegions::StdHexExp::v_IProductWRTBase_SumFacKernel
                              10.047  (11%)  dgemm_
                      8.796  (10%)  Nektar::MultiRegions::DisContField::v_AddTraceIntegral
                        8.796  (10%)  Nektar::MultiRegions::ExpList::v_IProductWRTBase
                          8.796  (10%)  Nektar::Collections::IProductWRTBase_IterPerExp::operator()
                            8.796  (10%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFac
                              8.796  (10%)  Nektar::StdRegions::StdQuadExp::v_IProductWRTBase_SumFacKernel
                                8.796  (10%)  dgemm_
                    2.732  (3%)  Nektar::MultiRegions::ExpList::v_BwdTrans_IterPerExp
          1.061  (1%)  Nektar::MultiRegions::ExpList::v_FwdTrans_IterPerExp
```

Figure 7: Call tree filtered by the dgemm BLAS function.