

# Scaling the Nektar++ Spectral/hp element framework to large clusters

A. Comerford, C. Cantwell, S.J. Clifford and S.J. Sherwin

Department of Aeronautics, Imperial College London, South Kensington Campus, SW7 2AZ, London, UK.

## Abstract

The present project was funded through a dCSE grant to improve the parallel performance of the open-source spectral/hp element framework, *Nektar++*, on HPC systems. Two particular tasks were undertaken: implementation of low energy basis preconditioning for substructure solvers and assess its parallel performance on the University clusters; and implementation of hybrid MPI/threading parallelism. Both these tasks have been successfully completed in *Nektar++*. The preconditioning strategies showed excellent performance; in particular, a fifteen-fold speed-up in solver performance was observed with the low energy basis preconditioning compared with the original implementation (prior to the dCSE funding). Additionally, the performance of the preconditioned conjugate gradient solver for the solution of the Navier-Stokes equations showed good scaling as the number of cores was increased (up to 432 cores) on Imperial College London based HPC systems. The initial implementation of the second task, to implement threading parallelism, has shown promise for future endeavours utilising this framework.

## 1 Overview

*Nektar++* is an open-source spectral/hp element framework ([www.nektar.info](http://www.nektar.info)) designed to allow the numerical solution of partial differential equations using low- and high-order finite element discretisations, and is a complete rewrite of the Nektar Incompressible Navier-Stokes flow solver. The framework supports both continuous and discontinuous finite element methods, multiple element types and is cross platform in nature.

This dCSE funded project was concerned with the extension of *Nektar++* onto large parallel systems. In particular two algorithmic extensions were implemented:

- Low-energy block preconditioning for substructure solvers in *Nektar++* with assessment of its parallel performance on the University clusters.
- Hybrid MPI/threading parallelism.

This report details the outcome of these algorithmic improvements to *Nektar++*. In section 2 the implementation and benchmarking of the low-energy block preconditioner is discussed, whilst in section 3 the implementation of hybrid MPI/threading parallelism is discussed.

## 2 Implementation of an effective $h$ and $p$ scaling preconditioner into the conjugate gradient solver.

Low-energy block preconditioning has been implemented into the conjugate gradient solver of *Nektar++*. In accordance with the proposal the preconditioner has been implemented to support multiple element types, that is hexahedral, tetrahedral and prismatic elements. The preconditioner has been thoroughly tested on multiple parallel systems based at Imperial College London. A detailed discussion of the theory, implementation and parallel performance is given in the following.

### 2.1 Preconditioning of the conjugate gradient method

In solving problems in engineering applications, such as the Navier-Stokes equations, it is required that the following elliptic boundary value problem is solved every time-step:

$$\nabla^2 u(\mathbf{x}) + \lambda u(\mathbf{x}) = f(\mathbf{x}) \quad (1)$$

defined on a domain  $\Omega$  of  $N_{el}$  non-overlapping elements. The standard spectral/hp approach is defined by,

$$u^\delta(\mathbf{x}) = \sum_{n=1}^{N_{dof}} \hat{u}_n \Phi_n(\mathbf{x}) = \sum_{e=1}^{N_{el}} \sum_{n=1}^{dim(V^\delta)} \hat{u}_n^e \phi_n^e(\mathbf{x}) \quad (2)$$

where  $N_{\text{dof}}$  is the number of degrees of freedom,  $\hat{u}_n$  are the global expansion coefficients,  $\phi_n(\mathbf{x})$  the polynomials defined in a space  $V^\delta$  of order  $P$ ,  $\hat{u}_n^e$  is the  $n^{\text{th}}$  local expansion coefficient within the element  $e$ . Approximating the solution of (2) and using a Galerkin discretisation of equation (1), i.e., find  $\mathbf{u}^\delta \in V^\delta$  such that

$$\mathcal{L}(v, u) = \int_{\Omega} \nabla v^\delta \cdot \nabla u^\delta + \lambda v^\delta u^\delta d\mathbf{x} = \int_{\Omega} v^\delta f d\mathbf{x} \quad \forall v^\delta \in V^\delta \quad (3)$$

This can be formulated in matrix terms as

$$\mathbf{H}\hat{\mathbf{u}} = \hat{\mathbf{f}} \quad (4)$$

where  $\mathbf{H}$  represents the Helmholtz matrix,  $\hat{\mathbf{u}}$  the unknown global coefficients and  $\mathbf{f}$  the inner product of the expansion basis with the forcing function. Additionally, a  $C^0$ -continuous expansion is obtained by decomposing the expansion basis into interior (having support in the interior of the element) and boundary modes (remaining modes on the element boundaries to make a complete expansion), and matching the boundary modes across the interface of the elements. The boundary modes can be further decomposed into vertex, edge and face modes, defined as follows: Vertex modes have support on a single vertex and the three adjacent edges and faces as well as the interior of the element; Edge modes have support on a single edge and two adjacent faces as well as the interior of the element; Face modes have support on a single face and the interior of the element. It is precisely this strong coupling between vertices, edges and faces that leads to a matrix of high condition number ( $\kappa$ ). Utilising the above-mentioned decomposition, we can write the matrix equation as:

$$\begin{bmatrix} \mathbf{H}_{bb} & \mathbf{H}_{bi} \\ \mathbf{H}_{ib} & \mathbf{H}_{ii} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}_b \\ \hat{\mathbf{u}}_i \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{f}}_b \\ \hat{\mathbf{f}}_i \end{bmatrix} \quad (5)$$

where the subscripts  $b$  and  $i$  are the boundary and interior degrees of freedom, respectively. This system then can be statically condensed allowing us to solve for the boundary and interior degrees of freedom in a decoupled manor. The statically condensed matrix is given by

$$\begin{bmatrix} \mathbf{H}_{bb} - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\mathbf{H}_{ib} & 0 \\ \mathbf{H}_{ib} & \mathbf{H}_{ii} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}_b \\ \hat{\mathbf{u}}_i \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{f}}_b - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\hat{\mathbf{f}}_i \\ \hat{\mathbf{f}}_i \end{bmatrix} \quad (6)$$

This is highly advantageous since by definition of our interior expansion this vanishes on the boundary hence  $\mathbf{H}_{ii}$  is block diagonal and hence can be easily inverted. The above sub-structuring has reduced our problem to solving the boundary problem:

$$\mathbf{S}_1\hat{\mathbf{u}} = \hat{\mathbf{f}}_1 \quad (7)$$

where  $\mathbf{S}_1 = \mathbf{H}_{bb} - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\mathbf{H}_{ib}$  and  $\hat{\mathbf{f}}_1 = \hat{\mathbf{f}}_b - \mathbf{H}_{bi}\mathbf{H}_{ii}^{-1}\hat{\mathbf{f}}_i$ . The decoupling of the boundary and interior degrees of freedom means the Schur complement matrix can be constructed at an elemental level. Although this new system typically has better convergence properties i.e lower  $\kappa$ , the system is still ill-conditioned leading to a convergence rate of the conjugate gradient (CG) routine that is prohibitively slow. For this reason we need to precondition  $\mathbf{S}_1$ . To do this an equivalent system is solved of the form:

$$\mathbf{M}^{-1}(\mathbf{S}_1\hat{\mathbf{u}} - \hat{\mathbf{f}}_1) = 0 \quad (8)$$

where the preconditioning matrix  $\mathbf{M}$  is such that  $\kappa(\mathbf{M}^{-1}\mathbf{S}_1)$  is less than  $\kappa(\mathbf{S}_1)$  and speeds up the convergence rate. Within the conjugate gradient routine the same preconditioner  $\mathbf{M}$  is applied to the residual vector  $\hat{\mathbf{r}}_{\mathbf{k}+1}$  of the CG routine every iteration:

$$\hat{\mathbf{z}}_{\mathbf{k}+1} = \mathbf{M}^{-1}\hat{\mathbf{r}}_{\mathbf{k}+1} \quad (9)$$

The key step is finding the appropriate choice of  $\mathbf{M}$  to speed up the conjugate gradient routine. For the spectral/ $hp$  element method care must be taken with respect to the scalability of  $h$  and  $p$ .

## 2.2 Preconditioners

In the following an appropriate preconditioner for the spectral/ $hp$  element method scaling with both  $h$  and  $p$  will be discussed.

### 2.2.1 Low Energy block preconditioning - p preconditioner

Low energy block preconditioning follows the methodology proposed by Sherwin & Casarin [2]. In this method a new basis is numerically constructed from the original basis, which allows the schur complement matrix to be preconditioned using a block preconditioner. The method is outlined briefly in the following.

Elementally it is possible to express the local approximation ( $\mathbf{u}^\delta$ ) as different expansions lying in the same discrete space ( $\mathbf{V}^\delta$ )

$$\mathbf{u}^\delta(\mathbf{x}) = \sum_{i=1}^{\dim(V^\delta)} \hat{u}_{1i} \phi_{1i}(x) = \sum_{i=1}^{\dim(V^\delta)} \hat{u}_{2i} \phi_{2j}(x) \quad (10)$$

Since both expansions lie in the same space it's possible to express one basis in terms of the other via a transformation, i.e.

$$\phi_2 = \mathbf{C}\phi_1 \implies \hat{\mathbf{u}}_1 = \mathbf{C}^T \hat{\mathbf{u}}_2 \quad (11)$$

Applying this to the Helmholtz operator it is possible to show that,

$$\mathbf{H}_2 = \mathbf{C}\mathbf{H}_1\mathbf{C}^T \quad (12)$$

For sub-structured matrices ( $\mathbf{S}$ ) the transformation matrix ( $\mathbf{C}$ ) becomes:

$$\mathbf{C} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \quad (13)$$

Hence the transformation in terms of the Schur complement matrices is:

$$\mathbf{S}_2 = \mathbf{R}\mathbf{S}_1\mathbf{R}^T \quad (14)$$

Typically the choice of expansion basis ( $\phi_1$ ) can lead to a Helmholtz matrix that has undesirable properties i.e poor condition number. By choosing a suitable transformation matrix ( $\mathbf{C}$ ) it is possible to construct a new basis, numerically, that is amenable to block diagonal preconditioning.

$$\mathbf{S}_1 = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{ve} & \mathbf{S}_{vf} \\ \mathbf{S}_{ve}^T & \mathbf{S}_{ee} & \mathbf{S}_{ef} \\ \mathbf{S}_{vf}^T & \mathbf{S}_{ef}^T & \mathbf{S}_{ff} \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{v,ef} \\ \mathbf{S}_{v,ef}^T & \mathbf{S}_{ef,ef} \end{bmatrix} \quad (15)$$

Applying the transformation ( $\mathbf{S}_2 = \mathbf{R}\mathbf{S}_1\mathbf{R}^T$ ) leads to the following matrix

$$\mathbf{S}_2 = \begin{bmatrix} \mathbf{S}_{vv} + \mathbf{R}_v \mathbf{S}_{v,ef}^T + \mathbf{S}_{v,ef} \mathbf{R}_v^T + \mathbf{R}_v \mathbf{S}_{ef,ef} \mathbf{R}_v^T & [\mathbf{S}_{v,ef} + \mathbf{R}_v \mathbf{S}_{ef,ef}] \mathbf{A}^T \\ \mathbf{A} [\mathbf{S}_{v,ef}^T + \mathbf{S}_{ef,ef} \mathbf{R}_v^T] & \mathbf{A} \mathbf{S}_{ef,ef} \mathbf{A}^T \end{bmatrix} \quad (16)$$

where ( $\mathbf{A}\mathbf{S}_{ef,ef}\mathbf{A}^T$ ) is given by

$$\mathbf{A}\mathbf{S}_{ef,ef}\mathbf{A}^T = \begin{bmatrix} \mathbf{S}_{ee} + \mathbf{R}_{ef} \mathbf{S}_{ef}^T + \mathbf{S}_{ef} \mathbf{R}_{ef}^T + \mathbf{R}_{ef} \mathbf{S}_{ff} \mathbf{R}_{ef}^T & \mathbf{S}_{ef} + \mathbf{R}_{ef} \mathbf{S}_{ff} \\ \mathbf{S}_{ef}^T + \mathbf{S}_{ff} \mathbf{R}_{ef}^T & \mathbf{S}_{ff} \end{bmatrix} \quad (17)$$

To orthogonalise the vertex-edge and vertex face modes, it can be seen from the above that

$$\mathbf{R}_{ef}^T = -\mathbf{S}_{ff}^{-1} \mathbf{S}_{ef}^T \quad (18)$$

and for the edge-face modes:

$$\mathbf{R}_v^T = -\mathbf{S}_{ef,ef}^{-1} \mathbf{S}_{v,ef}^T \quad (19)$$

Here it is important to consider the form of the expansion basis since the presence of  $(S_{ff}^{-1})$  will lead to a new basis which has support on all other faces; this is problematic when creating a  $C^0$  continuous global basis. To circumvent this problem when forming the new basis, the decoupling is only performed between a specific edge and the two adjacent faces in a symmetric standard region. Since the decoupling is performed in a rotationally symmetric standard region the basis does not take into account the jacobian mapping between the local element and global coordinates, hence the final expansion will not be completely orthogonal. Figure 1 demonstrates the elemental Schur complement matrix in the original basis and the new low energy basis, clearly the transformed Schur complement is more diagonally dominant.

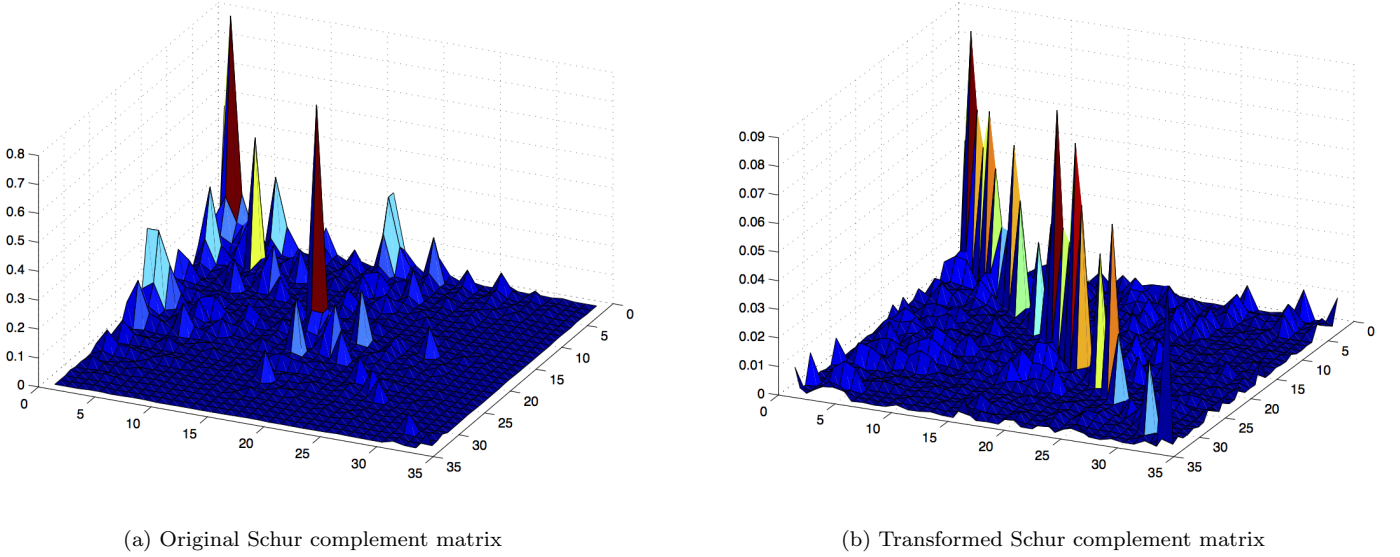


Figure 1: Comparison of Schur complement matrices in the two different bases. The low energy form of the Schur complement is more diagonally dominant.

The low energy basis creates a Schur complement matrix that although it is not completely orthogonal can be spectrally approximated by its block diagonal contribution. The final form of the  $p$  preconditioner is:

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{diag}[(\mathbf{S}_2)_{vv}] & 0 & 0 \\ 0 & (\mathbf{S}_2)_{eb} & 0 \\ 0 & 0 & (\mathbf{S}_2)_{fb} \end{bmatrix}^{-1} \quad (20)$$

where  $\mathbf{diag}[(\mathbf{S}_2)_{vv}]$  is the diagonal of the vertex modes,  $(\mathbf{S}_2)_{eb}$  and  $(\mathbf{S}_2)_{fb}$  are block diagonal matrices corresponding to coupling of an edge ( or face) with itself i.e ignoring the coupling to other edges and faces.

### 2.2.2 Coarse space preconditioning - h preconditioner

Preconditioning of the linear space is achieved by inverting only the vertex degrees of freedom (the linear finite element subspace) and has the form:

$$\mathbf{M}^{-1} = (\mathbf{S}_1^{-1})_{vv} \quad (21)$$

Since the mesh associated with higher order methods is relatively coarse compared with traditional finite element discretisations, the linear space can usually be directly inverted without memory issues. However such a methodology can be prohibitive on large parallel systems, due to a bottleneck in communication. Previously, for large parallel computations the preconditioning of the course space has been handled using ScaLapack [1]. In the present implementation this is handled using the  $XX^T$  library [3].  $XX^T$  is a parallel direct solver for problems of the form  $\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$  based around a sparse factorisation of the inverse of  $\mathbf{A}$ . To precondition utilising this methodology the linear sub-space is gathered from the expansion and the preconditioned residual within the CG routine is determined by solving:

$$(\mathbf{S}_1)_{\mathbf{v}\mathbf{v}}\hat{\mathbf{z}} = \hat{\mathbf{r}} \quad (22)$$

The preconditioned residual ( $\hat{\mathbf{z}}$ ) is then scattered back to the respective location in the global degrees of freedom.

### 2.2.3 h & p preconditioner

To combine the aforementioned  $h$  and  $p$  preconditioners an additive Schwarz preconditioner is utilised. This preconditioner is constructed by adding the two separate preconditioners i.e. the coarse space linear vertex block and the low energy block. To include this in the conjugate gradient routine a number of modifications must be made. Firstly, the system must be transformed to the low energy equivalent. Starting with the Schur complement system defined in the original basis:

$$\mathbf{S}_1\hat{\mathbf{u}} = \hat{\mathbf{f}} \quad (23)$$

and utilising the transformation equations (14) and (11) the following is obtained:

$$\mathbf{S}_2\hat{\mathbf{u}}_2 = \hat{\mathbf{f}}_2 \quad (24)$$

The above means that the conjugate gradient solver is in the low energy space; hence care must be taken when applying the different preconditioners i.e. the low energy basis preconditioner works on the low energy system whilst the linear space preconditioner works on the original system. This means when the residual vector ( $\hat{\mathbf{r}}$ ) is preconditioned the system must be transformed back to the original basis in order to precondition the linear degrees of freedom. This methodology is outlined in Figure 2. To summarise the low energy basis preconditioner is first applied to the residual vector and stored in  $\hat{\mathbf{z}}_{\text{LE}}$ . The non-preconditioned residual ( $\hat{\mathbf{r}}$ ) is then transformed from the low energy space to the original space. The linear space is then preconditioned using  $\mathbf{X}\mathbf{X}^T$ . The output ( $\hat{\mathbf{z}}_{\text{XXT}}$ ) is then transformed back to the low energy space. Finally the contributions of the two preconditioners are added together.

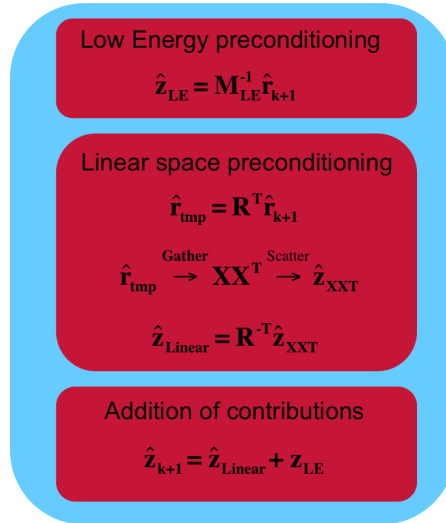


Figure 2: Strategy for applying both the low energy block and linear space preconditioner in the conjugate gradient routine.

## 2.3 Helmholtz cube - A simple serial example

The first benchmark is to demonstrate the performance of the LEBP (without linear space preconditioning) for a small serial simulation; this is primarily to demonstrate the effect of the  $p$  preconditioner on the convergence rate of the CG solver.

### Problem setup

The cube domain was discretised with of 6 tetrahedral elements. The following Helmholtz equation was considered:

$$\nabla^2 u(\mathbf{x}) + \lambda u(\mathbf{x}) = -(\lambda + 6)\sin(x)\sin(y)\sin(z) \quad (25)$$

where  $\nabla^2$  is the Laplacian and  $\lambda$  is a real positive constant. An initial condition of  $u = 0$  was used. For each preconditioning approach (No preconditioning, Diagonal preconditioning and Low energy basis preconditioning) the number of conjugate gradient iterations (CG) required to reach convergence was recorded. The simulations were performed for polynomial order of 3, 5, 7&9.

## Results

Figure 3 demonstrates the number of CG iterations to reach convergence for the aforementioned preconditioning strategies. It is evident that LEBP excels over the other approaches as the polynomial order increases; this is indicated by a reduced number of CG iterations. The primary reason for the large reduction is due to the decoupling of boundary modes, which improves the condition number of the Schur complement matrix.

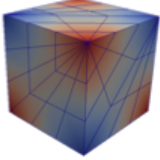
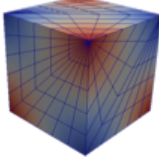
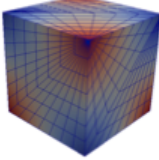
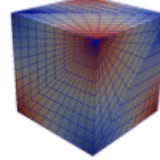
Result				
Order	P=3	P=5	P=7	P=9
Null	8	33	76	136
Diagonal	7	18	30	44
Low Energy	5	10	13	15
L <sub>2</sub> Error	0.041	0.0014	3.4e-05	5.6e-07

Figure 3: Performance of the Spectral/*hp* code *Nektar++* with LEBP. Discretisation: 6 tetrahedral elements with 3rd, 5th, 7th and 9th order polynomial approximation. Each entry represents the number of CG iterations to reach convergence for the different preconditioning strategies.

## 2.4 Implementation of the above preconditioner for use on massively parallel systems.

### 2.4.1 Cerebral aneurysm - A demonstration of mixed meshes and parallel scaling

As a second example the performance of the incompressible Navier-Stokes solver in a cerebral aneurysm (shown below with the pressure overlaid) for different preconditioning strategies is considered. This example demonstrates both the performance of the LEBP preconditioner on a mixed mesh as well as the scaling on a parallel system.

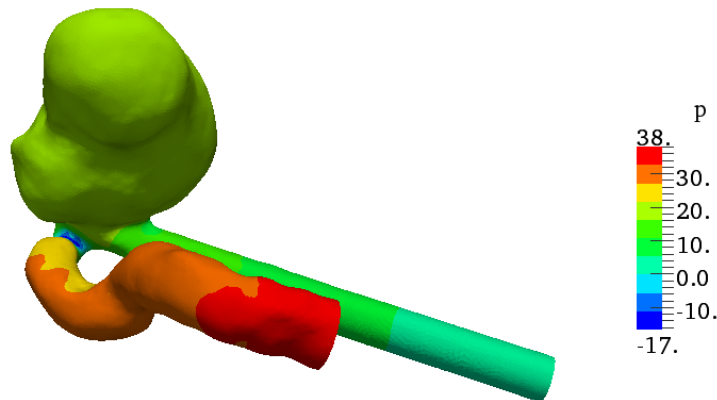


Figure 4: Cerebral aneurysm geometry with pressure contours overlain

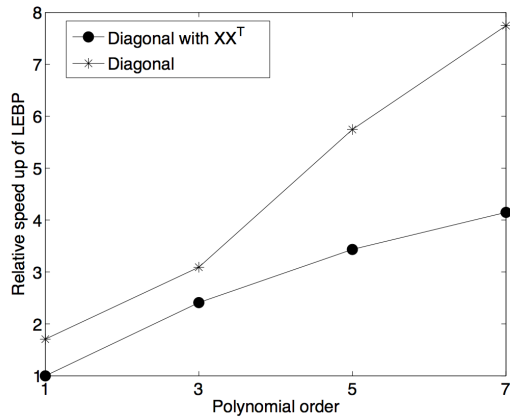
### 2.4.2 Problem setup

The solution of the Navier-Stokes was achieved utilising a splitting scheme, for more details see the description in the incompressible Navier-Stokes tutorial ([www.nektar.info](http://www.nektar.info)). In this approach four elliptic solves are required every time-step (one pressure

Poisson and three Helmholtz equations) which require preconditioning. The domain was discretised with  $\sim 6000$  prismatic and  $\sim 20000$  tetrahedral elements and an expansion order of 5 within each element. The solver performance was analysed for three different preconditioning strategies (LEBP, Diagonal and Diagonal with linear space preconditioning) at different polynomial order ( $p=1,3,5$  &  $7$ ). For each case, the solver was integrated in time for 110 time-steps and the mean time per time-step was calculated as the mean of 100 steps; the first ten steps were disregarded due to the effect of start-up transients. The comparisons were performed on the Imperial college HPC system cx1 (<http://www3.imperial.ac.uk/ict/services/hpc/highperformancecomputing>) using 72 cores.

### 2.4.3 Results

Figure 5 demonstrates the relative speed-up of low energy basis preconditioning for the aneurysm on 72 cores. From Figure 5(a) it is evident that as polynomial order increases there is a significant speed up over both diagonal preconditioning (labelled in Figure 5(a) as 'Diagonal') and diagonal with the linear space preconditioner (labelled in Figure 5(a) as 'Diagonal with  $XX^T$ '). In particular, at polynomial order 7 there is nearly an 8 times speed up over diagonal preconditioning and just over a 4 times with the addition of the linear space preconditioner.



(a) Relative speed up on 72 cores.

Solve	Pressure Poisson equation	Helmholz equation
Diagonal	678	139
Diagonal + $XX^T$	172	134
Low Energy + $XX^T$	45	10

(b) CG iterations per timestep.

Figure 5: Relative speed up and CG iteration count for low energy basis preconditioning compared with diagonal preconditioning with and without the linear space preconditioner  $XX^T$ . Discretisation:  $\sim 6000$  prismatic and  $\sim 20000$  tetrahedral elements with 5th order polynomial expansion.

To consider the comparison in more detail Figure 5(b) outlines the number of CG iterations required for the pressure Poisson and Helmholtz equations. There is approximately a fifteen-fold reduction in the number of CG pressure iterations between diagonal and low-energy preconditioning. This reduction is limited to a fourfold difference when the linear space preconditioner ( $XX^T$ ) is enabled. For the Helmholtz equation there is a 13.9 and 13.4 fold reduction going from diagonal to low energy and diagonal with the linear space to low energy, respectively.

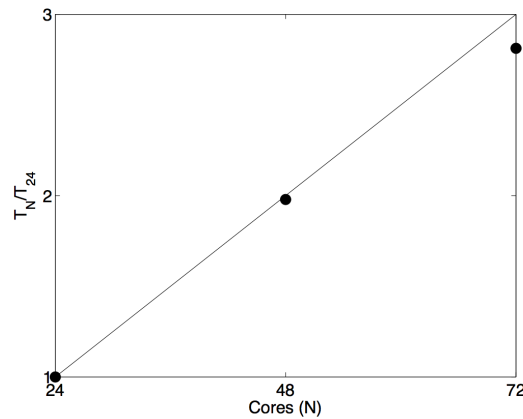


Figure 6: Parallel speed up of the Spectral/*hp* code *Nektar++* with low energy block preconditioning on cx1.

Figure 6 shows the parallel speed up (mean time per time-step compared with a reference time) of this example with the low

energy basis preconditioning enabled over a different number of cores: 24, 48 & 72. In the figure the reference time is taken as the time per time step for 24 cores. From the figure below the scaling is very close to ideal (solid black line) over the range of cores considered. Overall, with low energy basis preconditioning the solver seems to demonstrate very competitive performance. For example, for the 72 core scenario the time per time step is 1.2s ( $\Delta t = 0.001$ ).

## 2.5 Rabbit aorta - A demonstration on a large parallel system

In this section the speed up of the incompressible Navier-Stokes solver is analysed in a rabbit aorta (geometry and velocity field are shown in figure 8); this is a commonly used model for studying blood flow and the relationship with atherosclerosis. This example demonstrates the performance of low energy basis preconditioning on a large example and how this scales over many cores.

### 2.5.1 Problem setup

The domain consisted of  $\sim 20000$  prismatic and  $\sim 80000$  tetrahedral elements and an expansion order of 5 within each element. An example slice of the geometry and flow solution is shown below. As mentioned in the previous section the Navier-Stokes equations were solved using a splitting scheme. The solver was integrated in time for 110 time-steps and the mean time per time-step was calculated every ten steps (starting at step 10 in order to avoid any start-up transients). The simulations were performed on the Imperial College HPC system cx2 (<http://www3.imperial.ac.uk/ict/services/hpc/highperformancecomputing>) for three different core counts (192,288 & 432)

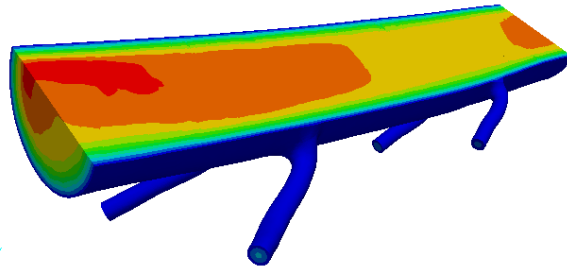


Figure 7: Aorta geometry used for benchmarking *Nektar++* on a large parallel system.

### 2.5.2 Results

Figure 8 demonstrates that the parallel speed-up of low energy basis preconditioning is rather good over the range of cores considered. In particular, for the maximum number of cores considered (432) the speed up is 2.05 (with the theoretical speed up being 2.25). Again, this example demonstrates very competitive performance of the incompressible Navier-Stokes solver with the time per time step at 432 cores being 0.7s ( $\Delta t = 0.001$ )

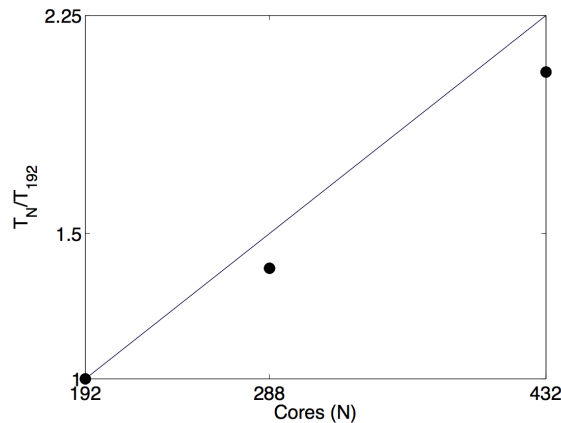


Figure 8: Parallel speed up of the Spectral/*hp* code *Nektar++* with low energy block preconditioning on cx2. Discretisation:  $\sim 20000$  prismatic and  $\sim 80000$  tetrahedral elements with 5th order polynomial expansion.



### 3 Implementation of hybrid MPI/threading parallelism.

MPI parallelism has multiple processes that exchange data using network or network-like communications. Each process retains its own memory space and cannot affect any other process's memory space except through the MPI API. A thread, on the other hand, is a separately scheduled set of instructions that still resides within a single process's memory space. Therefore threads can communicate with one another simply by directly altering the process's memory space. The project's goal was to attempt to utilise this difference to speed up communications in parallel code.

A design decision was made to add threading in an implementation independent fashion. This was achieved by using the standard factory methods which instantiate an abstract thread manager, which is then implemented by a concrete class. For the reference implementation it was decided to use the Boost library rather than native p-threads because *Nektar++* already depends on the Boost libraries, and Boost implements threading in terms of p-threads anyway.

It was decided that the best approach would be to use a thread pool. This resulted in the abstract classes `ThreadManager` and `ThreadJob`. `ThreadManager` is a singleton class and provides an interface for the *Nektar++* programmer to start, control, and interact with threads. `ThreadJob` has only one method, the virtual method `run()`. Subclasses of `ThreadJob` must override `run()` and provide a suitable constructor. Instances of these subclasses are then handed to the `ThreadManager` which dispatches them to the running threads. Many thousands of `ThreadJobs` may be queued up with the `ThreadManager` and strategies may be selected by which the running threads take jobs from the queue. Synchronisation methods are also provided within the `ThreadManager` such as `wait()`, which waits for the thread queue to become empty, and `hold()`, which pauses a thread that calls it until all the threads have called `hold()`. The API was thoroughly documented in *Nektar++*'s existing Javadoc style.

Classes were then written for a concrete implementation of `ThreadManager` using the Boost library. Boost has the advantage of being available on all *Nektar++*'s supported platforms. It would not be difficult, however, to implement `ThreadManager` using some other functionality, such as native p-threads.

Two approaches to utilising these thread classes were then investigated. The bottom-up approach identifies likely regions of the code for parallelisation, usually loops around a simple and independent operation. The top-down approach seeks to run as much of the code as is possible within a threaded environment.

The former approach was investigated first due to its ease of implementation. The operation chosen was the multiplication of a very large sparse block diagonal matrix with a vector, where the matrix is stored as its many smaller sub matrices. The original algorithm iterated over the sub matrices multiplying each by the vector and accumulating the result. The new parallel algorithm sends `ThreadJobs` consisting of batches of sub matrices to the thread pool. The worker threads pick up the `ThreadJobs` and iterate over the sub matrices in the job accumulating the result in a thread specific result vector. This latter detail helps to avoid the problem of cache ping-pong which is where multiple threads try to write to the same memory location, repeatedly invalidating one another's caches.

Clearly this approach will work best when the sub matrices are large and there are many of them. However, even for test cases that would be considered large it became clear that the code was still spending too much time in its scalar regions.

This led to the investigation of the top-down approach. Here the intent is to run as much of the code as possible in multiple threads. This is a much more complicated approach as it requires that the overall problem can be partitioned suitably, that a mechanism be available to exchange data between the threads, and that any code using shared resources be thread safe. As *Nektar++* already has MPI parallelism the first two requirements (data partitioning and exchange) are already largely met. However since MPI parallelism is implemented by having multiple independent processes that do not share memory space, global data in the *Nektar++* code, such as class static members or singleton instances, are now vulnerable to change by all the threads running in a process.

A new class, `ThreadComm`, was added to the existing communication class, `Comm`, of *Nektar++*. This class encapsulates a `Comm` object and provides extra functionality without altering the API of `Comm` (this is the Decorator pattern). To the rest of the *Nektar++* library this `Comm` object behaves the same whether it is a purely MPI `Comm` object or a hybrid threading plus MPI object. The existing data partitioning code can be used with very little modification and the parts of the *Nektar++* library that exchange data are unchanged. When a call is made to exchange data with other workers `ThreadComm` first has the master thread on each process (i.e. the first thread) use the encapsulated `Comm` object (typically an MPI object) to exchange the necessary data between the other processes, and then exchanges data with the local threads using direct memory to memory copies.

As an example: take the situation where there are two processes A and B, possibly running on different computers, each with two threads 1 and 2. A typical data exchange in *Nektar++* uses the `Comm` method `AllToAll(...)` in which each worker sends data to each of the other workers. Thread A1 will send data from itself and thread A2 via the embedded MPI `Comm` to thread

B1, receiving in turn data from threads B1 and B2. Each thread will then pick up the data it needs from the master thread on its process using direct memory to memory copies. Compared to the situation where there are four MPI processes the number of communications that actually pass over the network is reduced. Even MPI implementations that are clever enough to recognise when processes are on the same host must make a system call to transfer data between processes.

The code was then audited for situations where threads would be attempting to modify global data. Where possible such situations were refactored so that each thread has a copy of the global data. Where the original design of *Nektar++* did not permit this access to global data was mediated through locking and synchronisation. This latter approach is not favoured except for global data that is used infrequently because locking reduces concurrency.

The code has been tested on the Imperial College cluster cx1 and has shown good scaling. However it is not yet clear that the threading approach outperforms the MPI approach; it is possible that the speedups gained through avoiding network operations are lost due to locking and synchronisation issues. These losses could be mitigated through more in-depth refactoring of *Nektar++*.

## 4 Conclusion

With the aforementioned algorithmic implementations, *Nektar++* is now a modern spectral/hp element framework capable of solving a broad range of real-world engineering problems in an efficient manner. In particular, problems that are frequently encountered in many of the industrial and academic applications e.g. Formula One, bio-medical and acoustics, are now able to be solved efficiently using the framework.

## 5 Acknowledgements

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. <http://www.hector.ac.uk>

## References

- [1] L Grinberg, D Pekurovsky, S J Sherwin, and G.E. Karniadakis. Parallel performance of the coarse space linear vertex solver and low energy basis preconditioner for spectral/hp elements. *Parallel Computing*, 35:284–304, 2009.
- [2] Spencer J Sherwin and Mario Casarin. Low-energy basis preconditioning for elliptic substructured solvers based on unstructured spectral/hp element discretization. *Journal of Computational Physics*, 171:394–417, 2001.
- [3] Henry M Tufo and Paul F Fischer. Fast parallel direct solvers for coarse grid problems. *Journal of Parallel and Distributed Computing*, 61:151–177, 2001.