A parallel implementation of a finite element solver for statistical methods

Frydendall, Jan

Publication date: 2009

Document Version
Publisher's PDF, also known as Version of record

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A parallel implementation of a finite element solver for statistical methods

Jan Frydendall

July 21, 2009

DTU Informatics
Contents

1 Preliminary 3

2 Finite element implementation 4
   2.1 Galerkin finite element method ........................................ 4
   2.2 Advection-Diffusion ....................................................... 7

3 Sparse implementation 10
   3.1 The Storage formats ..................................................... 10
   3.2 The use of sparse matrices in the code .............................. 12

4 OpenMp implementation 15
   4.1 Construction of the OpenMP clauses .................................. 15

5 Performance of the OpenMP code 20
   5.1 Density of the finite element system matrix .......................... 21
   5.2 Parallel sparse direct solver .......................................... 22
   5.3 Performance results ..................................................... 23
   5.4 Conclusion ................................................................... 25

A The code 27
   A.1 Main program ................................................................. 27
   A.2 Sparse allocating ......................................................... 27
   A.3 Assembly .......................................................... 39
   A.4 Solver ............................................................ 52
   A.5 SuperLu interface ...................................................... 55
   A.6 Precision ............................................................ 55
1 Preliminary

The need for fast finite element solvers has long been a requirement for modern numerical simulation of large systems. Finite element methods is a very flexible method if the focus is on complex geometries in the domain, i.e. landscape modeling, fluid dynamics, material properties modeling etc. In the later years the finite element methods is also becoming known to statisticians who want to model more complex structures. A straight forward example is the modeling of the Fokker-Planck equation. The Fokker-Planck equation describes the time evolution of probability mass in a domain. The equation for the Fokker-Planck equation can be written as,

\[ \frac{\partial p}{\partial t} + \frac{\partial (a_i p)}{\partial x_i} - \frac{1}{2} \frac{\partial^2 (b^2_{ij} p)}{\partial x_i \partial x_j} = 0 \] (1)

The Fokker-Planck equation is a diffusion process where \( a(x, t) \) is the drift term and the \( b^2(x, t) \) is the diffusion term with continuous sample paths. The Fokker-Planck equation completely describes the time evolution of an stochastic process \( X_t \) once it is solved [3].

The structure of the Fokker-Planck equation is similar to the well known Advection-diffusion equation of fluid dynamics [6]

\[ \frac{\partial \psi}{\partial t} + \frac{\partial (u_i \psi)}{\partial x_i} - \mu \frac{\partial^2 (\psi)}{\partial x_i \partial x_j} = 0, \] (2)

where \( \psi \) is the concentration of the species that is convected/advected along the path of the velocity field \( U \) and dissipated with dispersion coefficient \( \mu \).

It should then be straight forward to make a finite element implementation of the Fokker-Planck equation since much of the literature is already well covered on the Advection-Diffusion equation. However, there are some very unpleasant Gibbs phenomena that can arise with the Advection part of the solver. In order to avoid these Gibbs phenomena an artificial diffusion has to be added to the Advection part of the solver. This is known in literature as the Streamline Upwind Petrov-Galerkin (SUPG) scheme. The introduction of the SUPG scheme will be covered in Section 2 together with the basic theory of the Advection-Diffusion scheme.

The report is build up in three parts. First the basic theory is covered of the finite element methods and the Fokker-planck and the Advection-Diffusion equations together with SUPG theory. Then the sequential implementation of the Finite element solver is covered. The solver will use the \( P_1 \) triangles, i.e. three point quadrature triangles. The implementation of the finite element solver will all be for unstructured grids for optimal flexibility. The unstructured implementation will also implicitly give solutions to structured triangle grids. The finite
element solver could with easy be extended to cover any type of geometrical structure in two
dimensional grids including \( P_2 \) triangles, \( Q_1 \) and \( Q_2 \) rectangles and extended to the third
dimension.

The second part of the report will cover the steps of introducing sparse matrices in the FOR-
TRAN solver. There are several books that covers the subject on implementing sparse matrices
in \( C \), however, it seems that one is left to ones own devices when it comes to implementing
sparse matrices in FORTRAN. In Section 3 a simple and efficient way is proposed.

Finally the report covers the steps that has to be taken in order to archive efficient parallel
performance from the finite element solver. The solver is implemented in FORTRAN and
run on SUN Solaris Sparc architecture machines. The sun company is on of the leading
developer of Symmetric Multi-Processors (SMP) computers and is on of the key players in the
introduction of the OpenMP standard. The SUN Performance Library (SPL), which is a part
of the SUN development studio, supports the OpenMP standard and no extra effort has to be
taken to make the SPL parallel.

The code is not designed for portability and therefore the code will be parallelized in order to
perform most efficiently on the SUN computers i.e. we will heavily make use of the SPL.

2 Finite element implementation

This section is not a completely introduction to the theory behind finite elements method
nor the theory to Advection-Diffusion or the Fokker-Planck. The section acts only as a small
introduction to the theory which should make the understanding of the solver easier.

In this report we assume that we have an incompressible fluid i.e. \( \partial_x u = 0 \), therefore the
Advection diffusion equation simplifies to

\[
\frac{\partial \psi}{\partial t} + u_i \frac{\partial (\psi)}{\partial x_i} - \mu \frac{\partial^2 (\psi)}{\partial x_i \partial x_j} = 0, \tag{3}
\]

2.1 Galerkin finite element method

The finite element method used in this rapport is the Galerkin finite element method. The
method is very flexible once the code is developed. One can easily change to higher order
method without changing the code. We only have to change the element mapping and the
quadrature.
Complex geometries put together from arbitrary triangles can be quite tedious to derive the derivatives from such elements. However, if we can find a mapping from a standard triangle where all derivatives are made easy to obtain the derivatives on the complex triangle. The idea is to find a mapping from a reference element in a fix coordinate system and then calculate the mapping functions. In this report a very brief introduction is given without the proper mathematical rigor. Consider the reference $P_1$ element (the right-angled triangle) shown in Figure 2.1 together with a arbitrary element from solution space. The idea is to calculate the derivatives on the reference element and map the derivatives to the elements in the mesh.

In the given element $P_1$ there are three nodes where the local derivatives can be obtained. The derivatives can found by interpolation the three nodes on the element with linear polynomials. The linear polynomials are defined as:

\[
\phi_1(r, s) = 1 - r - s \\
\phi_2(r, s) = r \\
\phi_3(r, s) = s
\]

With the above definition the $\psi$ function can now be written in local coordinates:

\[
\psi(r, s) = \sum_{i=1}^{3} \phi_i \psi_i
\]
mates $x_i$ and $y_i$ at the nodal points the mapping in the reference element is given as

$$x(r, s) = \sum_{i=1}^{3} \phi_i x_i$$

$$y(r, s) = \sum_{i=1}^{3} \phi_i y_i$$

Using the chain rule the Jacobian is:

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial r} \frac{\partial r}{\partial x} + \frac{\partial}{\partial s} \frac{\partial s}{\partial x}$$

$$\frac{\partial}{\partial y} = \frac{\partial}{\partial r} \frac{\partial r}{\partial y} + \frac{\partial}{\partial s} \frac{\partial s}{\partial y},$$

collecting the terms in matrix form:

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \end{bmatrix},$$

where $J$ is defined as:

$$J = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} \end{bmatrix}$$

Multiplying (3) with the arbitrary test function $\bar{\psi}$, the following expression is obtained, written out in fully coordinates:

$$\int_{\Omega} \bar{\psi} \left[ u \frac{\partial \psi}{\partial x} + v \frac{\partial \psi}{\partial y} - \mu \frac{\partial^2 \psi}{\partial x^2} - \mu \frac{\partial^2 \psi}{\partial y^2} \right] dxdy = 0,$$

for the diffusion process the divergence and Green’s theorems are used where the boundary integral is set to zero, thus,

$$\int_{\Omega} \left[ \bar{\psi} u \frac{\partial \psi}{\partial x} + \bar{\psi} v \frac{\partial \psi}{\partial y} \right] dxdy - \int_{\Omega} \mu \left[ \frac{\partial \psi}{\partial x} \frac{\partial \bar{\psi}}{\partial x} + \frac{\partial \psi}{\partial y} \frac{\partial \bar{\psi}}{\partial y} \right] dxdy = 0,$$

Now it is a simple matter of finding the integration mapping by using the chain rule (8) the final equation becomes

$$\int_{\Omega} \left[ \bar{\psi} u \frac{\partial \psi}{\partial r} + \bar{\psi} v \frac{\partial \psi}{\partial s} \right] drds - \int_{\Omega} \mu \left[ \frac{\partial \psi}{\partial r} \frac{\partial \bar{\psi}}{\partial r} J^{-1} + \frac{\partial \psi}{\partial s} \frac{\partial \bar{\psi}}{\partial s} J^{-1} \right] drds = 0.$$
2.2 Advection-Diffusion

The Advection-Diffusion partial differential equation is a hyperbolic differential equation which are notoriously difficult to solve numerically. If a front is advected along the underlining current then Gibbs phenomena will arise around the front. Gibbs phenomena are small oscillations around the edges of sharp gradients, like that of a front. Depending on the system these oscillations will in worst case eventually deteriorate or destroy the solution. In Figures (2a - 2b) and (2c - 2d) the solution to the Advection-diffusion equation is shown. The solution is a cone that is transported along a counter-clockwise rotating velocity field. At time step $t = 15$ the Gibbs phenomena are already visible and these small oscillations multiplies as the solution time increases. At time step $t = 60$ the entire domain is covered with small oscillations if the solution is integrated even further then the small oscillations would eventually destroy the solution.

![Figure 2: A cone is advected along in counter-clock vise rotating velocity field. The cone is shown at four different times $t = \{0, 15, 45, 60\}$. The solution field has many small oscillations around the cone.](image)

To this point there are only two known solution to this problem, if the solution is an Eulerian solution. The Gibbs phenomena can be suppressed with a filter or a limiter [4]. The other option is to introduce a balancing (artificial) diffusion to blur out the Gibbs phenomena. The inspiration for the balancing diffusion is taken from the finite difference approach of the upwind scheme.

The key idea is to implement the balancing diffusion in the direction of the resulting velocity
Figure 2: A cone is advected along in counter-clockwise rotating velocity field. The cone is shown at four different times $t = \{0, 15, 45, 60\}$. The solution field has many small oscillations around the cone.

field. This will of course make the balancing diffusion anisotropic. By introducing a weighting function as [6]

$$\psi_a = \psi_a + \alpha \psi^*$$

$$= \psi_a + \frac{\alpha h}{2} \frac{U_i}{|U|} \frac{\partial \psi_a}{\partial x_i},$$

where $\alpha$ is quantity that has to be calculated for each element.

$$\alpha = \coth Pe - \frac{1}{Pe},$$

where

$$Pe = \frac{|U|h}{2k},$$

where $|U| = \sqrt{U_i U_i}$. $Pe$ is called the Peclet number and is a dimensionless quantity that relates the rate of the advection of a flow to the diffusion. The quantity $h$ is a reasonable defineable element size. There is no real definition of how to find $h$, it is just some form of element measure [6]. In this case it is taken as the diameter of the element. This is done by finding the diameter of the inscribed circle of the triangle. In the book by [6] the choice of $h$ is chosen such that the direction of $h$ coincides with the velocity vector $U$. This is not implemented for the unstructured mesh, however, for the structured mesh this is implemented in the code. The equation (13) is designed such that the balancing diffusion is only active in the direction of the flow, in the other direction the balancing diffusion should be zero.
If eq. (13) is substitute into the advection part of (3) then we will get an additional diffusion term:

\[
\int_{\Omega} \left[ \bar{\psi} \frac{\partial \psi}{\partial x} + \bar{\psi} \frac{\partial \psi}{\partial y} \right] dxdy + \int_{\Omega} \left[ u \bar{\psi} \frac{\partial^2 \psi}{\partial x^2} + v \bar{\psi} \frac{\partial^2 \psi}{\partial y^2} \right] dxdy + \int_{\Omega} \left[ u \bar{\psi} \frac{\partial^2 \psi}{\partial x \partial y} + v \bar{\psi} \frac{\partial^2 \psi}{\partial y \partial x} \right] dxdy
\]

(16)

(17)

The two last terms are the balancing diffusion that suppresses the buildup of oscillations around fronts in the solver. However, the balancing can not suppress all oscillations and small oscillation may still appear in the solution. However, the oscillations are now so small that they can be neglected for the most parts. We will not go deeper into the theory of suppressing oscillations in this report. The above algorithm is implemented into the subroutine \texttt{Peclet}.

We have not applied the chain rule for eq. (16) as this a trivial matter.

In Figure 3 the Streamline upwind Petrov-Galerkin method is applied to the same condition as in Figures (3a - 3b) (3c - 3d). However, the SUPG diffusion is clearly suppressing the Gibbs phenomena such that the solution is preserved. There is no indications in the figures that there are any visible small oscillations.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{As in Figures (2a - 2b). With the SUPG scheme the solution is no longer prone to the Gibbs phenomena in the same extent as before.}
\end{figure}
3 Sparse implementation

In order to save memory and to have a very fast code the sparsity of the finite element solver will be exploited.

3.1 The Storage formats

Choosing a sparse storage system on the other hand is more difficult. There are quite a few available. The sparskit [5] supports up to 16 different storage schemes. The different schemes are listed below.

- DNS Dense format
- BND Linpack Banded format
- CSR Compressed Sparse Row format
- CSC Compressed Sparse Column format
- COO Coordinate format
- ELL Ellpack-Itpack generalized diagonal format

Figure 3: As in Figure (2c - 2d). With the SUPG scheme the solution is no longer prone to the Gibbs phenomena in the same extent as before.
• DIA Diagonal format
• BSR Block Sparse Row format
• MSR Modified Compressed Sparse Row format
• SSK Symmetric Skyline format
• NSK Nonsymmetric Skyline format
• LNK Linked list storage format
• JAD The Jagged Diagonal format
• SSS The Symmetric Sparse Skyline format
• USS The Unsymmetric Sparse Skyline format
• VBR Variable Block Row format

The storage schemes used in this report will be the COO and CSC storage format. The coordinate format or more popularly the triplet is by fare the easiest to work with in the assembly phase, however, it requires more memory the other more condense sparse storage schemes.

We will assemble the matrixes in the triplet format and then convert the triplet format to the CSC format before the matrices are passed to the direct solver. As a small remark, the sparse matrix structure found in matlab is also the same as the one we have chosen. Matlab uses the triplet format for communication with the user and the CSC format for all mathematical operations done behind the scenes. The triplet format is contain in two integer vectors $ia$ and $ja$ with respectively the $i$th and $j$th coordinate and one real valued vector $a$ which stores the coordinate value $k_{ij}$. The CSC storage format is like the triplet format also made up from two integer vectors $ia$ and $ja$ containing the coordinates and one real values vector $a$. However, the CSC format uses a more sophisticated memory efficient storage format for the coordinate vector. A small example is given to show the difference between the two storage formats. Given the matrix below the triplet format can straight forward be deducted: There are 13 non-zero elements in the matrix (18).

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
2 & 6 & 0 & 0 & 9 \\
3 & 0 & 7 & 0 & 0 \\
4 & 0 & 0 & 8 & 0 \\
5 & 0 & 0 & 0 & 10 \\
\end{pmatrix}
\]
3 SPARSE IMPLEMENTATION

The CSC storage format for the matrix (18)

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ia</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ja</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The CSC storage format content is similar to the triplet format it contains the same values in the real vector $a$ and in the row integer vector $ia$. However, in the column integer vector $ja$ we now only store the pointer of each column in the vectors value and row. Thus the content of $ja(i)$ is the position in arrays $a$ and $ia$ where the $i$-th row starts [5].

The CSC storage format is well covered in literature [5, 2] and how to convert to CSC from triplet will not derived here. The sparskit toolbox [5] and the Suitesparse of [2] are respectively a FORTRAN and C library that can do these operations efficiently.

3.2 The use of sparse matrices in the code

The naive implementation of the sparse matrix will be to assemble the finite element matrices in full form and then convert them to sparse matrices once there are assembled. This approach is, however, not very efficient when we have very large data sets. With the finite element matrices in sparse form we can solve them with a standard direct sparse solver. The direct solver will covered in the next section.

However, if we want to have a fast an efficient code then we need to store the finite element matrices in sparse format from the beginning. There are a few standard libraries which can allocate the matrices in sparse format. The CS sparse toolkit from [2] is a C library that that can work with matrices in sparse form. The CS sparse toolkit is written to flawlessly interact with matlab. The code from CS sparse toolkit could be translated to FORTRAN or we could link the C codes directly to the FORTRAN libraries through the compiler with the respective CS sparse libraries. However, the latter can be very hazardous because of the difference in the memory structure of C and FORTRAN. To our knowledge there is no standard FORTRAN toolkit that can allocate the sparse matrices directly from within the heart of the code. It
seems that if one wants to implement a sparse structure in FORTRAN one has to start from scratch.

The first thing that is needed for sparse structure is a way of allocating the memory for the triplet structure. We need a tool to count the non-zero entries and to allocate the coordinate vectors $ia$ and $ja$. When we have allocated the triplet coordinate vectors we can use them to lookup any value in the finite element matrix. This can be done with a binary search algorithm that will find the right position in the triplet format given a coordinate pair $(i,j)$ from the given finite element matrix. The coordinate pair indicates the position of the real value of the finite element matrix entry. Once the position is located in the triplet vectors the real value is stored in the triplet value vector $a$ at the right position. Thus, we can assemble the finite element matrices directly into the sparse structure. It is not very difficult to look up the non-zero entries if we first have the coordinate vectors and the coordinate pair. However, the difficult part arises when we want to allocate the $ia$ and $ja$ coordinate vectors. If we had a structured fix mesh we could once for all determined the non-zero entries by assemble the matrices in full format once and then convert to there sparse structure and count the non-zero entries and register the coordinate vectors.

However, we would like to have an adaptable unstructured finite element solver, therefore we have to find a way of counting the non-zero entries and a method to allocate the $ia$ and $ja$ coordinate vectors. The easiest way, but not the cleverest way, is to assemble one matrix in the preparation phase and then convert to sparse structure to get the desired information. This will work for any structure. However, this approach is very slow and will in the end take more time then the actually solving of the sparse system.

The efficient way is to count the nodal coordinates in the mesh that is used by the solver. This can be done if we have information on the adjacent nodal points in the mesh. This information is found by the subroutine $connect$ in the code. The connect code is a matlab algorithm from the book [4]. The algorithm makes an element table of the mesh. The element table is a table with information of adjacent elements in the Element to Element EToE and the Element to Faces and EToF. The latter will not be used in the used in the code. For more information on the algorithm consult the book. The next step is to count the nodal points in the mesh. This is done in the code $allocate__nnz$ if we have the three vertices in one triangle we can count the number of adjacent points that are connected to the respectively points. In Figure 3.2 a small structured mesh is shown. The nodal points are numbered with blue and the elements are marked with red numbers. If we look at node 25 in Table (1) we can see that the node is adjacent to the following nodal points $18, 19, 24, 25, 26, 31, 32$ including it self. So
**Figure 4:** A small structured mesh with nodal points number with blue and the element number with red

<table>
<thead>
<tr>
<th>Node</th>
<th>Adjacent points</th>
<th>Number of points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 8, 1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1, 2, 3, 8, 9</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>18, 19, 24, 25, 26, 31, 32</td>
<td>7</td>
</tr>
<tr>
<td>41</td>
<td>36, 37, 43, 44</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 1:** A adjacent table with four nodal points which shows how the structured mesh is connected

every nodal point that is away from the boundary can at most be adjacent to seven points. In Table 1) the connectivity of the adjacent points are shown for strategic points in the mesh. Points on the boundary can have between 3 and 5 points.

To find the actual number of non-zeros is done by counting the number of adjacent points in the mesh. Start with one and then add the number of adjacent points for every node in the mesh.

The allocation of the triplet coordinate vectors is a bit trickier. Once the total number of non-zeros is known we can actual loop through the same algorithm again to store the coordinates in \( ia \) and \( ja \). For a mesh consisting of three nodal points per triangle element we can deduct that for every element we use up to seven points in the integration of the element. Therefore it is a matter of book keeping storing the coordinates. If we look at Table (1) we can write
the coordinates for node 1 as (1, 2), (1, 8), (1, 1) and for node 2 (2, 1), (2, 2), (2, 3), (2, 8), (2, 9). This is done the algorithm \texttt{allocate\_RowsCols}. We finally have to find a lookup table to store arbitrary coordinates \((i, j)\) with the \(a_{ij}\). Once we have allocated the \(ia\) and \(ja\) we can use a binary search to lookup the corresponding \(a_{ij}\) in the mesh. This is done in the algorithm \texttt{lookup}.

The above algorithms \texttt{allocate\_nnz} and \texttt{allocate\_RowsCols} are inspired by the technical report [1] and in the code found on John Burkardt homepage. The software on the homepage is free under the GNU LGPL license. I have made my own interpretation of the code so they suit my purposes. In my search for appropriated sparse implementation in FORTRAN his homepage was the only place where I was able to find something about sparse implementation of finite element methods in Fortran. As mention before there several ways of doing this in C and C++.

4 OpenMp implementation

To facilitate the code for high performance capability of modern SMP computer we have to consider a parallel implementation of the code. We will use the OpenMP FORTRAN API for the parallelization of the code. The choice for OpenMP over MPI is because we want the code to be portable to a least Solaris computers with the SUN STUDIO compilers. The OpenMP API enables us to still maintain the code on computers without multithread support and we don’t have to rewritten the entire code to get good performance. In theory OpenMP API can be implemented into the code without any modifications to the existing code. However, in practice it sometimes necessary to rewrite some parts of the code in order to get good performance.

Before we start on the implementation we have to stress out that we will not be able to get linear scaling with the entire code. There are too many synchronization in the subroutines. However, there is still a substantial amount of performance to get from the parallelization of the code and we aspect to get very good performance with the assembly of the system matrix and the direct solver.

4.1 Construction of the OpenMP clauses

The OpenMP standard backbone is the \texttt{!$omp do} clause. Since much of scientific computer codes is about crunching huge matrices in for/do loops. This clause will be used intensively
throughout the code. The parallelism archive with the standard !$omp do clause will not be described in this report.

The main program first calls the subroutine read\_mesh. This subroutine reads in the mesh from the supplied mesh files, i.e. the mesh vertices and mesh elements. We can not make this code run in parallel since the file can only be accessed sequential in FORTRAN. The next subroutine called is the allocate\_sparse this is a small routine that allocates the sparse triplet format. This subroutine calls a set of new subroutines to calculate the connectivities of the elements connect and the assessment of the non-zeros allocate\_nnz and finally the allocation of the rows and columns of spares triplet format allocate\_RowsCols. All three subroutines have been made parallel and the implementation was straight forward. In the regions where the algorithms are counting we have used the !$omp critical clause for the synchronizing of the threads.

However, in the subroutine allocate\_RowsCols there is a special case where we normally could not use an omp do clause. This is because the code segment is sequential. When we do the !$omp do in the code shown in the Figure 5 the connectivity table is divided into $n$ number of threads segments. Each of these segments are then calculated in the do loop. However, this would give erroneous results because we are allocating the coordinates for row and columns vectors. The counting of the coordinates is no longer in an ordered form. However, we are saved in the end of the program, because we have to sort the vectors anyway to have a faster look up table for the binary search algorithm. Therefore we can after all make an efficient parallel subroutine without an extra overhead for a sort algorithm.

After the allocation of the triplet sparse coordinate vectors the driver subroutine is called. In this subroutine the actual assembly of the matrices is performed.

Since the assembly is a four nested loops that runs over a set of dense sub matrices. However, we have to be careful since in a finite element assembly we actually sum the elements that are occurring more then once. Therefore we have to consider the reduction clause. In FORTRAN we have the possibility of using the reduction clause for arrays as for scalars. In C and C++ we can only use the Reduction clause for scalars. The reduction clause will put the appropriated OpenMp critical clauses around the summation of the elements. A snip of the code from the subroutine assembleConT3 can be seen in the Figure 6. The code segment also calls the binary search function lookup\_k we have not parallelized this function since it is called from each parallel segment.

The reduction clause should be very efficient with arrays in FORTRAN. However, this im-
When executing this algorithm we can not be sure that \texttt{adjcopy} is count up as in the sequential case. However, this will not influence the result of the subroutine. At the end of the subroutine we do sort the triplet, so the end result does match that of the sequential algorithm.

\begin{verbatim}
do i=1,mesh%nel
  j=EToE(i,:)==i
  if(any(j)) EToE(i,pack((/1,2,3/),mask=j))=-1
  t=EToE(i,:)
  e=mesh%EToV(i,:)
  do k=1,3
    if( i<t(k) .or. t(k)<0) then
      !$omp critical
      triplet%row(adjcopy(e(l(k,:)))) = e(l(k,:))
      triplet%col(adjcopy(e(l(k,:)))) = e(lm(k,:))
      adjcopy(e(l(k,:))) = adjcopy(e(l(k,:))) + (/1,1/)
      !$omp end critical
    end if
  end do
end do
!

$omp end do
$omp workshare
tempsort=triplet%row*expnns+triplet%col
p=/(i,i=1,triplet%nnz)/
p1=/(i,i=1,mesh%nnd*mesh%nnd*mesh%nel)/
p2=/(i,i=1,mesh%nnd*mesh%nel)/
$omp end workshare nowait
$omp single
call sortv(tempsort,1,p)
$omp end single
$omp workshare
triplet%col=triplet%col(p)
$omp end workshare nowait
$omp end parallel
\end{verbatim}

\textbf{Figure 5}: A section of the \texttt{Allocate\_RowsCols} code with the sort trick
implementation had the opposite effect in this code. In Table (2) we see the timings from the assemble code. There is a speedup from 1 to 2 threads and again from 2 to 4 threads. Increasing the number of threads from 4 to 8 the speedup starts to decrease. At 64 threads the code segment actually takes longer to be executed then it would in the sequential case.

This is due to the fact that the reduction schedule has to synchronize all threads at the end of the loop. If the values, as in this case, are scatter over the arrays then each thread has to wait for each other to be synchronized. This generates a huge overhead. The solution to this problem is divided the domain into n-thread sub-domains. In each sub-domain the matrices can be assemble without the use of the reduction schedule.

To decompose the domain into n-thread sub-domains we will have to store the array elements that will be used in the assembly of the matrices. From the Figure 6 we can see that ival is the index that has to be reduced in the OpenMP section. In the subroutine Allocate_RowsCols we have stored all the variables \text{\{iel, krow, kcol, ival\}} into an array \text{par\%matrix} and sorted the indexes after the \text{ival} index. The same \text{ival} index will maximum occur 9 times in the

\begin{verbatim}
!$omp parallel private(iel,krow,ii,kcol,jj,ival) default(shared)
!$omp & reduction(+: K,C,A,f)
!$omp do
    do iel=1,mesh%nel
        do krow=1,nnd
            ii=mesh%EToV(iel,krow)
            do kcol=1,nnd
                jj=mesh%EToV(iel,kcol)
                call lookup_k(triplet%row, triplet%col, ii, jj,ival)
                K(ival) = K(ival) + ke(iel,krow,kcol)
                C(ival) = C(ival) + ce(iel,krow,kcol)
                A(ival) = A(ival) + ae(iel,krow,kcol)
            end do
            f(ii) = f(ii) + fe(iel,krow)
        end do
    end do
!$omp end do nowait
!$omp end parallel$
\end{verbatim}

Figure 6: A section of the assemble code with the reduction cluse
Table 2: Timings from the reduction assemble code

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.3</td>
</tr>
<tr>
<td>2</td>
<td>29.1</td>
</tr>
<tr>
<td>4</td>
<td>20.9</td>
</tr>
<tr>
<td>8</td>
<td>21.1</td>
</tr>
<tr>
<td>16</td>
<td>30.0</td>
</tr>
<tr>
<td>32</td>
<td>54.1</td>
</tr>
<tr>
<td>64</td>
<td>192.0</td>
</tr>
</tbody>
</table>

entire array. The trick is to divided the domain into n-thread sub-domains by partitioning the ival in the par%matrix such that there are no overlappings in ival sequences, e.g. if we have a sequences of numbers \{21 21 22 22 22\} then we have to make sure that the partition is made between \{21|22\} and not between \{22|22\}. With this partition we can assemble the code without the need of a reduction schedule. In Figure 7 the OpenMP loop is demonstrated.

From the Figure 7 we also see that we have made the same partition for the right hand side vector f. The code for the partitioning can be found in the Appendix.

On all other constructions will use the OpenMP workshare clause which is also exclusive to FORTRAN. The workshare clause supports the semantics of the FORTRAN 90 array syntax structure, since much of the code is written the FORTRAN 90 array syntax. In the subroutine Peclet all the array calculation is done with FORTRAN 90 array structure. A snip from the code can be seen in Figure 8.

The effect of the parallel code executed on four threads can be seen in Figure 4.1. The only part that is not running on more then one thread is the read_mesh subroutine. From the figure we can see that all threads are doing work, however, there are some small section in the time line which some or all threads are idle. This is due to the switch between the different subroutines in the code. There is really nothing to be done in these parts, however, this will of course add time to the overall timing of the code.

With the tuning of the code for parallel operation we are now ready to test the code and to measure the improvements obtained from the parallelization. This will be described in the next section.
Figure 7: A section of the new assembly matrix code. The code uses a partition scheme to implement the summation of the dense matrices into the global matrices

5 Performance of the OpenMP code

To test the potential performance that can be archived with the parallel code we will test the code with various threads numbers. The setup of the solver will be based on a large mesh and with a constant rotating velocity field. The rotating velocity field will ensure us that cone the adverting cone stays inside the domain under the test. The rotation test is also classic test for the Advection algorithm. We will not come into the details on the rotation test in the rapport since it is not the main topic. In the test we assemble the finite element matrices and solve it for 50 time steps. We will do this on a set of different threads \{1, 2, 4, 8, 16, 32\} to find the limit of the speedup that can be archived with the code. All the performance will be evaluated on the SUN-HPC Euler server. We have used the SUN Grid Engine to submit the jobs to the Grid Engine.
5 PERFORMANCE OF THE OPENMP CODE

! Calculate the Peclet number
! computation of element Peclet number (at the centroid)
!$omp workshare
nl12=sqrt((x(:,1)-x(:,2))**2.0_dp+(y(:,1)-y(:,2))**2.0_dp)
nl13=sqrt((x(:,1)-x(:,3))**2.0_dp+(y(:,1)-y(:,3))**2.0_dp)
nl23=sqrt((x(:,2)-x(:,3))**2.0_dp+(y(:,2)-y(:,3))**2.0_dp)
sper=(nl12+nl13+nl23)/2.0_dp
flow_l2 = sqrt(flowx(:) * flowx(:) + flowy(:) * flowy(:))
area=sqrt(sper*(sper-nl12)*(sper-nl13)*(sper-nl23))
!$omp end workshare
if(all(flowx==0.0_dp)) then
  flow_h=nl23
else if( all(flowy==0.0_dp)) then
  flow_h=nl13
else
!$omp workshare
  flow_h=area/sper
!$omp end workshare
end if

Figure 8: A section of the subroutine Peclet code with the workshare clause

5.1 Density of the finite element system matrix

The mesh that we have selected for the test has 263169 nodes i.e. the size of the system matrices would be $263169^2 = 69257922561$, and the mesh consist of 524288 elements. The number of non-zeros in system is 1838081 and the density of the system matrix is $\mu = \frac{1838081}{263169^2} \cdot 100 = 0.0027\%$. For a two dimensional system the large mesh is almost unrealistically large, however, to obtain some results that are measureable we had to go with the large mesh. For a smaller and more realistic mesh with 13000 elements the solver is incredibly fast and the timings are not reliable. However, the code could easily be modified to deal with three dimensions. If we have a mesh of 13000 elements in the plane and we want to discretizes the vertical into 15 layers then the system matrix is close to the size of the larger mesh.
5 PERFORMANCE OF THE OPENMP CODE

5.2 Parallel sparse direct solver

The solver that we have used until now in the sequential code is the direct solver provided by the SUN Performance Library (SPL). The direct solver `dgssfs` is the one-in-all interface for the direct solver. There are currently two flavors to chose from, the first is the `SP-solve` which is a FORTRAN solver with a factorization step. The other solver is `c`-based `SuperLU` direct solver. When using the `SuperLU` from a FORTRAN program we have to remember that `c`-arrays are zero-based and FORTRAN is one-based. Therefore we have to convert the triplet vectors to csc zero based. This is done with the subroutine `coo_to_csc0`.

One thing that is odd with SPL is that in the documentations is stated that the `SP-solve` should be parallel. However, I have tried in many different ways to unlock this feature. However, it seems that this feature is not present in the current SUN Studio 12. It has been in previously installments of the SUN Studio series. Hopefully it will become a part of the SPL in later installments of the SUN Studio.

Since we are not able to use the build in direct solver from the SPL, we have to use another solver. We will use the `SuperLU_MT` where MT is short for Multi-thread. The `SuperLU_MT` is free software that can be downloaded from the authors’ homepage. The `SuperLU_MT` supports posix threads and pthreads, Solaris threads and OpenMP. However, we have not been able to get the OpenMP support to work. We therefore compile the `SuperLU_MT` to support Solaris threads. From an architectural point-of-view there is no difference between Solaris threads and OpenMP threads. It is just a matter of book keeping; and of course the easy OpenMP interface is substitute with the interface of the Solaris threads which is more complicated. However, since the `SuperLU_MT` is already put together we don’t need to be worry about that fact. The `SuperLU_MT` interface is compiled and a small FORTRAN module is created to control the `SuperLU_MT`. In the newest version of the `SuperLU_MT` we are also given the choice of `colamd` [2] reordering of the matrices. This should be a more efficient than the ordinary `mmd` reordering. If we only want to use the sequential version of the code we suggest to use the `SuperLU` from the SPL, since it is
more optimizes towards the Solaris platforms.

5.3 Performance results

We have measured the performance at several instances in the code. Firstly we have measured the time the code is spending in the allocation part of the code. Secondly we have measured the time spend in `assembleConT3` and in the `Adv_SUPGT3` subroutines. Thirdly we have measured the overall assemble section which consists of the `assembleConT3` and in the `Adv_SUPGT3` and other small subroutine calls. Fourthly we have measured the performance of the `SuperLU` at every time step.

![Figure 10: (10a) wall-clock times for all the subroutines in the code. (10b) wall-clock times for the two subroutines assembleConT3 and the Adv_SUPGT3](image)

In Figure 10a the wall-clock time is shown for the execution of the code on the different numbers of threads: \{1, 2, 4, 8, 16, 32\}. From the Figure 10a it is clear that the parallelism archived with OpenMP implementation is very good going from 1 thread to 2. In the beginning we see a super-scaling for all measured segments with expectance from the `Allocate` segment. We cannot hope to see any real scaling behavior from the `Allocate` segment throughout the test. This is due to the large overhead generating by multiple calls to many different small subroutines in this segment. Going from 2 to 4 threads we have linear scaling of the code segments which can be seen from Figure 10d. However, this efficiently cannot be maintained when the number of threads are increased to 8 and 16. We still see scalability, however, the efficiently is decreasing as the overhead from the OpenMP clauses are increases. There is only a minimal effect going to 32 threads. If the number of threads would be increased above 32 we will see that the measured times start to increase instead of decreasing.
In this test we have only used a constant wind field for the simulations. However, if we have had a time dependence wind field we would have had to assemble the system matrix at every time step. Therefore it is important to have a fast and scalable assembled code. In the Figures 10a 10b and 10d we see that the overall assemble segment is responding very well to the parallel implementation. Decomposing the Allocation segment into the two main components assembleConT3 and Adv_SUPGT3 we see that these two subroutines scales very good compared to the number threads used in the execution of the code. In section 4.1 we used much effort to get the assemble code to scale. The reduction clause where the easiest to implement put proved to scale very badly after 4 threads. First after rewriting the assemble algorithm did we get a algorithm that would scale beyond 4 threads i.e. Figure 7. However, looking at Figure 10c we see that the overall time is used by the direct solver. Therefore it is important that the solver scales good with the number of threads used in the execution of the code. The SuperLU_MT solver behaves very good in the range from 2 to 16 threads which can be seen from the Figure 10a, here we have only shown the mean of 49 time steps iteration. The overall time used by the code is shown in Figure 10c this figure is created by adding the different segments to a total time (there are some problems with the measuring at the HPC center at this time). This means that the SuperLU_MT solver is called 49 times and this segments is the most dominating part of the execution time. The choice for the SuperLU_MT solver was that we wanted a clean multi-thread direct solver. There are of course many other solvers on the market. If we wanted the absolutely the best performance we should consider a hybrid between OpenMP and MPI.
5 PERFORMANCE OF THE OPENMP CODE

5.4 Conclusion

The conclusion to the OpenMP implementation of the code is that we have archived an overall execution time that is gone done by a factor 8 which can be seen from the Figure 10d if we use 16 threads. This is a very good performance and this is best that we can hope for with an OpenMP implementation. If we wanted to have a better performance on the overall execution time we should consider MPI or hybrid code between OpenMP and MPI. However, on of strengths with OpenMP is that our code still can be used without OpenMP support in the compiler. This gives a very portable structure which we cannot have with MPI. In a MPI implementation we would have had to totally reconstruct the code segments to support MPI. This is very difficult and very time consuming. However, if the MPI code would have to be used extensively over a very long time period then the investment in the construction of the MPI code would be advantageous.

To round off the OpenMP implementation in this report we have calculated the parallel efficiency of the code segments. The efficiency can be calculated from \( E = \frac{T_1}{T_p} \), where \( p \) is number of threads, \( E \) is the efficiency, \( T_1 \) is execution time for on thread and \( T_p \) is the execution time for \( p \) threads. For linear scaling the efficiency should be \( E = 1 \) and super scaling above \( E > 1 \) and scaling below linear scaling \( E < 1 \). The efficiency gives an indication on the payoff of the number threads invested in the code. The Figure 11 shows the efficiency of the parallelization. The efficiency goes below 0.5 after 16 threads after this there is no longer any payoff regarding the execution times of the code segments. Instead could the extra computational power be used for other programs. A special note on the Allocation segment which seems to performance very badly in all the test. In this special segment there are many calls to different subroutines. Some of these subroutines scales very well and other do not.

![Figure 11: The efficiency for the parallel code segments as a function of the thread count.](image-url)
We have not wanted to show how each small subroutine scales in this report because is not very interesting. If someone should ever comes to use this report and the code here in we recommended to investigate the scalability of this section to convince themselves that about the scalability of the many small subroutines in this segment.

References


A THE CODE

A The code

A.1 Main program

program main
  use precision
  use precondition, only : read_mesh
  use sparse_precondition
  use solver
  implicit none
  real(dp),parameter :: theta=1.0_dp,CFL=0.9_dp,time=10.0_dp
  integer :: nns,nel,neg,nnz,steps,t,npar
  type(mesh_matrix) :: mesh
  type(COO_matrix) :: triplet
  type(par_matrix) :: par
  real(dp),dimension(1:2) :: viscosity
  real(dp) :: t1,t2
  character :: cpu*2
  call getarg(1,cpu)
  read(cpu,'(i2.2)')par%npar
  print *,’Number of threads: ’,par%npar
  ! Call in the preconditioners from the subroutines.
  ! Allocate the arrays for the system
  t1 = omp_get_wtime()
  call read_mesh(mesh)
  !print *,’Mesh okay’
  ! Find the size of non-zero elements
  ! call allocate_sparse(par,mesh,triplet)
  !print *,’Allocation okay’
  ! Allocate the arrays
  viscosity=0.03_dp
  call driver(par,mesh,triplet,viscosity,CFL,theta,time)
  t2 = omp_get_wtime()
  print *,’Total time: ’,t2-t1
end program main

A.2 Sparse allocating

! File: sparse_precondition.f90
! Author: jf
! Created on January 3, 2009, 12:48 PM
!
! The subroutines are modified version from John Burkardt online library
! https://people.scs.fsu.edu/~burkardt/f_src/f_src.html
! All credit to John Burkardt
! The subroutines have been modified to suit my purpose.

MODULE sparse_precondition

use utils, only : print_matrixI
use precision
use omp_lib
use sunperf
use m_inssor
contains

subroutine allocate_sparse(par,mesh,triplet)
  implicit none
  ! Find the number of nonzero elements from the geometri of the mesh.
  !
  type(mesh_matrix),intent(inout) :: mesh
  type(COO_matrix),intent(inout) :: triplet
  type(par_matrix),intent(inout) :: par
  ! local variables
  integer, allocatable, dimension (:) :: colsum
  real(dp) :: t1,t2,t3
  !
  ! Determine the element neighbor array, just so we can estimate
  ! the nonzeros.
  !
  t3=omp_get_wtime()
call omp_set_num_threads(par%npar)
  !t1=omp_get_wtime()
allocate (colsum(1:mesh%nns+1) )
call connect(mesh)
  !t2=omp_get_wtime();print *,'connect: ',t2-t1
  !print *,'Connect Okay'
  !t1=omp_get_wtime()
call allocate_nnz(mesh,colsum,triplet)
  !t2=omp_get_wtime();print *,'Allocate_nnz: ',t2-t1
  !print *,' Number of nonzeros :
  !t1=omp_get_wtime()
call allocate_RowsCols(par,mesh,colsum,triplet)
  !t2=omp_get_wtime();print *,'Allocate_RowsCols: ',t2-t1
  !print *,'Allocation of Row and Cols okay'
  !t1=omp_get_wtime()
call boundary_node(mesh)
!t2=omp_get_wtime();print *, 'Boundary Node: ', t2-t1
!t2=omp_get_wtime()
print *, 'Allocate time: ', t2-t3
!
! Set up the sparse row and column index vectors.
!
end subroutine allocate_sparse

subroutine connect(mesh)
  implicit none
  type(mesh_matrix), intent(inout) :: mesh

! local variables
  integer, dimension(1:3*mesh%nel,1:4) :: spNodeToNode
  integer, dimension(1:3*mesh%nel,1:2) :: fnodes
  integer, dimension(1:3*mesh%nel) :: id,p,EToEv,EToFv
  integer, allocatable, dimension(:,:) :: matchL,matchR
  integer, allocatable, dimension(:) :: imatch
  integer :: Nfaces,Nnodes,i,icount,nel
  logical, dimension(1:3*mesh%nel-1) :: indices
allocate(mesh%EToE(1:mesh%nel,1:3),mesh%EToF(1:mesh%nel,1:3))
nel=mesh%nel
Nfaces=3;
Nnodes = maxval(mesh%EToV)
!
! create list of all faces 1, then 2, & 3
!$omp parallel private(i) default(shared)
!$omp workshare
fnodes(1:nel,1:2)=mesh%EToV(:,(/1,2/))
fnodes(nel+1:2*nel,1:2)=mesh%EToV(:,(/2,3/))
fnodes(2*nel+1:3*nel,1:2)=mesh%EToV(:,(/3,1/))
!$omp end workshare nowait
!$omp do do i=1,3*nel
  call sort(fnodes(i,:))
  call inssor(fnodes(i,:))
  fnodes(i,:)=fnodes(i,:)-1
end do
!$omp end do nowait
!
! set up default element to element and Element to faces connectivity
!$omp workshare
mesh%EToE = spread((/(i,i=1,nel)/),nel,2)
mesh%EToF=transpose(spread((/(i,i=1,3)/),2,nel))
! uniquely number each set of three faces by their node numbers
  
  \[ id = \text{fnodes}(:,1) \cdot N\text{nodes} + \text{fnodes}(:,2) + 1 \]
  
  \[ \text{spNodeToNode}(::,1) = id \]
  
  \[ \text{spNodeToNode}(::,2) = (i, i=1, nel*3) / (i, i=1, nel*3) / \]
  
  \[ \text{spNodeToNode}(::,3) = \text{reshape}(\text{mesh}\%\text{EToE}, (/3*nel/)) \]
  
  \[ \text{spNodeToNode}(::,4) = \text{reshape}(\text{mesh}\%\text{ETO F }, (/3*nel/)) \]
  
  \[ p = ((i, i=1, nel*3)) / \]
  
  !$omp end workshare nowait
  
  !$omp end parallel

! Now we sort by global face number.
  
  call sortv(\text{spNodeToNode}(::,1), 1, p)
  
  !$omp parallel
  
  !$omp workshare
  
  \text{spNodeToNode}(::,2:4) = \text{spNodeToNode}(p, 2:4)
  
  !$omp end workshare nowait
  
  !$omp end parallel
  
  !call print\_matrixI(\text{spNodeToNode}); stop

! find matches in the sorted face list
  
  !$omp parallel
  
  !$omp workshare
  
  indices = \text{spNodeToNode}(1:3*nel-1,1) == \text{spNodeToNode}(2:3*nel,1)
  
  icount = count(indices)
  
  !$omp end workshare nowait
  
  !$omp end parallel

! make links reflexive
  
  allocate(\text{matchL}(1:2*icount, 1:4), \text{matchR}(1:2*icount, 1:4), \text{imatch}(1:icount))
  
  !$omp parallel
  
  !$omp workshare
  
  matchL = 0; matchR = 0
  
  imatch = pack((i, i=1, 3*nel-1), mask=indices)
  
  matchL(1:icount,:) = \text{spNodeToNode}(imatch,:) \]
  
  matchL(icount+1:2*icount,:) = \text{spNodeToNode}(imatch+1, :)
  
  matchR(1:icount,:) = \text{spNodeToNode}(imatch+1, :)
  
  matchR(icount+1:2*icount,:) = \text{spNodeToNode}(imatch, :)

! insert matches
  
  \text{ETOEv} = \text{reshape}(\text{mesh}\%\text{EToE}, (/3*nel/)) \]
  
  \text{ETO V} = \text{reshape}(\text{mesh}\%\text{ETO F}, (/3*nel/))
  
  \text{ETO Ev}(\text{matchL}(::,2)) = \text{matchR}(::,3); \text{ETO Fv}(\text{matchL}(::,2)) = \text{matchR}(::,4)
  
  \text{mesh}\%\text{ETO E} = \text{reshape}(\text{ETO Ev}, (/nel, 3/)) \]
  
  \text{mesh}\%\text{ETO F} = \text{reshape}(\text{ETO Fv}, (/nel, 3/))
  
  !$omp end workshare nowait
!$omp end parallel
deallocate(matchL,matchR,imatch)

end subroutine connect

subroutine allocate_nnz(mesh,colsum,triplet)
  implicit none
  type(mesh_matrix),intent(in) :: mesh
  type(COO_matrix),intent(out) :: triplet
  integer,intent(out),dimension(:) :: colsum
  !local variables
  integer :: i,k,npar
  integer,dimension(1:mesh%nns) :: nz
  integer,dimension(1:mesh%nel,1:3) :: EToE
  integer,dimension(1:3,1:2) :: l
  integer,dimension(1:3) :: t
  logical,dimension(1:3) :: j
  real(dp) :: t1,t2
  !$omp parallel
  !$omp workshare
  EToE=mesh%EToE
  !$omp end workshare nowait
  !$omp single
  npar=omp_get_num_threads()
  !$omp end single
  !$omp end parallel
  nz=1
  l(1,:)=(/1, 2/)
  l(2,:)=(/2, 3/)
  l(3,:)=(/3, 1/)
  !$omp parallel if(npar>4) num_threads(4) private(i,k,j,t) default(shared)
  !$omp do
  do i=1,mesh%nel
    j=EToE(i,:)==i
    if(any(j)) EToE(i,pack(/1,2,3/,mask=j))=-1
    t=EToE(i,:)
    do k=1,3
      if( i<t(k) .or. t(k)<0) then
        !$omp critical
        nz(mesh%EToV(i,l(k,:)))=nz(mesh%EToV(i,l(k,:)))+1
        !$omp end critical
      end if
    end do
  end do
  end do
!!$omp end do nowait  
!!$omp end parallel  
colsum=cumsumI((/1,nz/))

triplet%nnz=colsum(mesh%nns+1)-1

dend subroutine allocate_nnz

subroutine allocate_RowsCols(par,mesh,colsum,triplet)
  implicit none
  type(mesh_matrix),intent(in) :: mesh
  type(COO_matrix),intent(inout) :: triplet
  type(par_matrix),intent(inout) :: par
  integer,intent(in),dimension(:) :: colsum
  !local variables
  integer :: i,k,node,nnz,iel,krow,kcol,ii,jj,icount
  real(dp) :: expnns
  integer,dimension(1:mesh%nel,1:3) :: EToE
  integer,dimension(1:mesh%nns) :: adjcopy
  integer,dimension(1:3,1:2) :: l,lm
  real(dp),dimension(1:triplet%nnz) :: tempsort
  integer,dimension(1:triplet%nnz) :: p
  integer,dimension(1:mesh%nnd*mesh%nel) :: p1
  integer,dimension(1:mesh%nnd*mesh%nel) :: p2
  integer,dimension(1:3) :: t,e
  logical,dimension(1:3) :: j
  nnz=triplet%nnz
  allocate(triplet%row(1:nnz),triplet%col(1:nnz),triplet%values(1:nnz),par%vector(1:mesh%nel*mesh%nnd),par%matrix(1:mesh%nnd*mesh%nel,1:mesh%nnd*mesh%nel),par%startblock(1:par%npar,1:2),par%endblock(1:par%npar,1:2))
  EToE=mesh%EToE
  triplet%row=-1
  triplet%col=-1
  adjcopy=colsum(1:mesh%nns)
  expnns=10.0_dp**ceiling(log10(real(mesh%nns)))
  !$omp parallel if(par%npar>4) num_threads(4) private(node) default(shared)
  !$omp do
  do node = 1, mesh%nns
    !$omp critical
    triplet%row(adjcopy(node)) = node
    triplet%col(adjcopy(node)) = node
    adjcopy(node) = adjcopy(node) + 1
    !$omp end critical
  end do
  !$omp end do nowait
!$omp end parallel
l(1,:)=(/1, 2/);lm(1,:)=(/2, 1/
1(2,:)=(/2, 3/);lm(2,:)=(/3, 2/
1(3,:)=(/3, 1/);lm(3,:)=(/1, 3/
!$omp parallel if(par%nparr>4) num_threads(4) private(i,k,j,t,e) default(shared)
!$omp do
! When executing this algorithm we can not be sure that adjcopy is
! count up as in the sequential case. However, this will not
! influence the result of the subroutine. At the end of the subroutine
! we do sort the triplet, so the end result does match
! that of the sequential algorithm.
do i=1,mesh%nel
    j=EToE(i,:)==i
    if(any(j)) EToE(i,pack(/1,2,3/),mask=j)=-1
    t=EToE(i,:)
    e=mesh%EToV(i,:)
do k=1,3
    if( i<t(k) .or. t(k)<0) then
        !$omp critical
        triplet%row(adjcopy(e(l(k,:)))) = e(l(k,:))
        triplet%col(adjcopy(e(l(k,:)))) = e(lm(k,:))
        adjcopy(e(l(k,:))) = adjcopy(e(l(k,:))) + (/1,1/
        !$omp end critical
    end if
end do
!$omp end do
!$omp workshare
tempsort=triplet%row*expnns+triplet%col
p=/(i,i=1,triplet%nnz)/
p1=/(i,i=1,mesh%nnd*mesh%nnd*mesh%nel)/
p2=/(i,i=1,mesh%nnd*mesh%nel)/
!$omp end workshare nowait
!$omp single
call sortv(tempsort,1,p)
!$omp end single
!$omp workshare
triplet%col=triplet%col(p)
!$omp end workshare nowait

!$omp do private(iel,krow,ii,kcol,jj,k,icount)
do iel=1,mesh%nel
    do krow=1,mesh%nnd
        ii=mesh%EToV(iel,krow)
do kcol=1,mesh%nnd
  jj=mesh%EToV(iel,kcol)
  call lookup_k(triplet%row, triplet%col, ii, jj,k)
  icount=(iel-1)*mesh%nnd*mesh%nnd+(krow-1)*mesh%nnd+kcol
  par%matrix(icount,1:4)=(/iel,krow,kcol,k/)
end do
par%vector((iel-1)*mesh%nnd+krow,1:3)=(/iel,krow,ii/)
end do
end do

!$end do nowait
!$omp single
  call sortv(par%matrix(:,4),1,p1)
  call sortv(par%vector(:,3),1,p2)
!$omp end single
!$omp workshare
  par%matrix(:,1:3)=par%matrix(p1,1:3)
  par%vector(:,1:2)=par%vector(p2,1:2)
!$omp end workshare nowait
!$omp end parallel

par%startblock=0
par%endblock=0
do i=1,par%npar
  ! Finding the start and end positions in the decomposed mesh.
  ! First in the matrix structure
  par%startblock(i,1)=par%startblock(i,1)+size(par%matrix,1)/(par%npar)*(i-1)+1
  par%endblock(i,1)=par%endblock(i,1)+size(par%matrix,1)/(par%npar)*i
  ! Next in the vector structure
  par%startblock(i,2)=par%startblock(i,2)+size(par%vector,1)/(par%npar)*(i-1)+1
  par%endblock(i,2)=par%endblock(i,2)+size(par%vector,1)/(par%npar)*i
  ! If the partition is in between two same numbers then move the end point the left to get the
  ! to be between two different to each other numbers. i.e.
  ! 20 21 22 22
  ! |              }
  ! new partition
  ! 20 21 22 22
  ! |              }
  ! this domain decomposition will avoid the reduction clause in the assemble phase.
  !
  ! First allocate the matrix
  if(par%matrix(par%endblock(i,1),4)==par%matrix(par%endblock(i,1)-1,4).and.i<par%npar) then
    icount=count(par%matrix(par%endblock(i,1)-10:par%endblock(i,1)-1,4)==par%matrix(par%endblock(i,1),4))+1
    par%endblock(i,1)=par%endblock(i,1)-icount
  !print *,par%matrix(par%endblock(i,1)-10:par%endblock(i,1)+2,4)
  !par%startblock(i+1,1)=-icount
THE CODE

end if
! Allocating the vector
if(par\vector(par\endblock(i,2),3)==par\vector(par\endblock(i,2)-1,3).and.i<par\npar) then
  icount=count(par\vector(par\endblock(i,2)-10:par\endblock(i,2)-1,3)==par\vector(par\endblock(i,2),3))
  par\endblock(i,2)=par\endblock(i,2)-icount
! print *,par\vector(par\endblock(i,2)-10:par\endblock(i,2)-1,3)
par\startblock(i+1,2)=-icount
end if
end do
!
print *,par\matrix(par\startblock(:,1),4)
!
print *,par\matrix(par\endblock(:,1),4)
end subroutine allocate_RowsCols

subroutine boundary_node(mesh)
implicit none
type(mesh\_matrix),intent(inout) :: mesh
! Local variables
integer :: i
logical,dimension(1:3) :: j
allocate(mesh\BCTYPE(1:mesh\nns),mesh\BC(1:mesh\nns))
!$omp parallel default(shared)
!$omp workshare
mesh\BC=.false.;mesh\BCTYPE = 1
!$omp end workshare nowait
!$omp do private(i,j)
do i=1,mesh\nel
  j=mesh\EToE(i,:)==i
  if(any(j)) then
    mesh\BC(mesh\EToV(i,pack((/1,2,3/),mask=j)))=.true.
    mesh\BCTYPE(mesh\EToV(i,pack((/1,2,3/),mask=j)))=2
  end if
end do
!$omp end do nowait
!$omp end parallel
end subroutine boundary_node

subroutine dirichletBC( mesh, triplet, A, C,f )
implicit none
type(mesh\_matrix),intent(in) :: mesh
type(COO\_matrix),intent(in) :: triplet
real(dp),intent(inout),dimension(:) :: A,C,f
! Local Variables
integer, parameter :: DIRICHLET = 2
integer, dimension(1:triplet%nnz) :: ia, ja
integer :: i, node, column, nnz
integer, dimension(1:size(mesh%BCTYPE)) :: BCTYPE
nnz = triplet%nnz; ia = triplet%row; ja = triplet%col; BCTYPE = mesh%BCTYPE
 !$omp parallel default(__auto)
 !$omp do
 do i = 1, nnz
  node = ia(i)
  if ( BCTYPE(node) == DIRICHLET ) then
   column = ja(i)
   if ( column == node ) then
    A(i) = 1.0_dp
    C(i) = 1.0_dp
    f(node) = 0.0_dp!node_bc(node)
   else
    A(i) = 0.0_dp
    C(i) = 0.0_dp
   end if
  end if
 end do
 !$omp end do nowait
 !$omp end parallel
end subroutine dirichletBC

subroutine lookup_k(ia, ja, i, j, k)
 implicit none
 integer, intent(in) :: i, j
 integer, intent(in), target, dimension(:) :: ia, ja
 integer, intent(out) :: k
 ! Local Variables
 integer, pointer :: p(:)
 integer :: mid, offset, l
 p => ia
 k = 0
 offset = 0
 l=0
 do while (size(p) > 0)
  mid = size(p)/2+1; l=offset+mid
  if (ia(l) < i .or. (ia(l) == i .and. ja(l) < j)) then
   p => p(mid+1:)
   offset = offset + mid
  else if (ia(l) > i .or. (ia(l) == i .and. ja(l) > j)) then
   p => p(:mid-1)
  else
k = offset + mid  ! SUCCESS!
    return
  end if
end do
if(size(p) == 0) k = -1
end subroutine lookup_k

function cumsumI(x)
  implicit none
  integer :: i, n
  integer, intent(in), dimension(:) :: x
  integer, allocatable, dimension(:) :: cumsumI
  n = size(x, 1)
  allocate(cumsumI(1:n))
  cumsumI = 0
  cumsumI(1) = x(1)
!$omp parallel private(i) shared(cumsumI,x,n)
!$omp do ordered
  do i = 2, n
    !$omp ordered
    cumsumI(i) = cumsumI(i-1) + x(i)
    !$omp end ordered
  end do
!$omp end do
!$omp end parallel
end function cumsumI

subroutine coo_to_csc0(ia, ja, a, csc)
  implicit none
  integer, intent(in), dimension(:) :: ia, ja
  real(dp), intent(in), dimension(:) :: a
  type(csc_matrix), intent(inout) :: csc
!local variables
  integer :: n, k, p, m
  integer, dimension(size(ia)) :: ja1, ia1
  integer, allocatable, dimension(:) :: w, T
  n = size(ia)
  allocate(w(0:n))
  w = 0; T = 0
  do k = 1, n
    w(ja(k)) = w(ja(k)) + 1
  end do
  m = count(w > 0) + 1
  allocate(csc%colptr(1:m), csc%rowind(1:n), csc%values(1:n), T(0:m-1))
A THE CODE

```fortran
subroutine coo_to_csc0
  implicit none
  integer,intent(in),dimension(:) :: ia, ja
  real(dp),intent(in),dimension(:) :: a
  type(csc_matrix), intent(inout) :: csc
  !local variables
  integer :: n, k, p, m
  integer, allocatable, dimension(:) :: w, T
  n=size(ia)
  allocate(w(0:n))
  w=0; T=0
  do k=1, n
    w(ja(k))=w(ja(k))+1
  end do
  m=count(w>0)+1
  allocate(csc%colptr(1:m),csc%rowind(1:n),csc%values(1:n),T(1:m))
  csc%nnz=n; csc%n=m-1
  csc%colptr(1:m)=cumsumI(w(0:m-1))+1
  T=csc%colptr
  do k=1, n
    p=T(ja(k))+1
    T(ja(k))=T(ja(k))+1
    csc%rowind(p)=ia(k)
    csc%values(p)=a(k)
  end do
end subroutine coo_to_csc0

subroutine coo_to_csc1(ia, ja, a, csc)
  implicit none
  integer, intent(in), dimension(:) :: ia, ja
  real(dp), intent(in), dimension(:) :: a
  type(csc_matrix), intent(inout) :: csc
  !local variables
  integer :: n, k, p, m
  integer, allocatable, dimension(:) :: w, T
  n=size(ia)
  allocate(w(0:n))
  w=0; T=0
  do k=1, n
    w(ja(k))=w(ja(k))+1
  end do
  m=count(w>0)+1
  allocate(csc%colptr(1:m),csc%rowind(1:n),csc%values(1:n),T(1:m))
  csc%nnz=n; csc%n=m-1
  csc%colptr(1:m)=cumsumI(w(0:m-1))+1
  T=csc%colptr
  do k=1, n
    p=T(ja(k))
    T(ja(k))=T(ja(k))+1
    csc%rowind(p)=ia(k)
    csc%values(p)=a(k)
  end do
end subroutine coo_to_csc1

subroutine csc1_vector_mult(Ap, Ai, A, x, b)
  ! multiply A * x = b
  integer, dimension(:), intent(in) :: Ai, Ap
  real(dp), dimension(:), intent(in) :: A, x
  real(dp), dimension(:), intent(out) :: b
```

real(dp) :: Aij
integer :: n,i,p,j

n = size(Ap)-1
! Initialize b
b = 0.0_dp
! b = A * x
!$omp parallel default(__auto)
!$omp do
do j = 1,n
  do p = Ap(j), Ap(j+1) - 1
    i = Ai(p)
    Aij = A(p)
    b(i) = b(i) + Aij * x(j)
  enddo
enddo
!$omp end do nowait
!$omp end parallel
end subroutine csc1_vector_mult

END MODULE sparse_precondition

A.3 Assembly

! File: precondition.f90
! Author: jf
!
! Created on January 3, 2009, 12:48 PM
!
module precondition
  use utils
  use precision
  use sparse_precondition
  use omp_lib
  !use sunperf
  contains
  subroutine read_mesh(mesh)
    implicit none
    type(mesh_matrix),intent(inout) :: mesh
    !local Variables
    integer :: i,j,k
    type(mesh_matrix) :: mesh

open(0100,file='mesh.xy.bac',status='old')
open(0200,file='mesh.elmtab.bac',status='old')
read(0100,*) j, mesh%nns
read(0200,*) k, mesh%nel
mesh%nnd=k
allocate(mesh%vx(1:mesh%nns),mesh%vy(1:mesh%nns),mesh%EToV(1:mesh%nel,1:3))
doi=1,mesh%nns
read(0100,*)mesh%vx(i),mesh%vy(i)
end do
do i=1,mesh%nel
read(0200,*)mesh%EToV(i,1:k)
end do
close(0100)
close(0200)
end subroutine read_mesh

!! The Galerkin Way

subroutine dtscale(mesh,x,y,CFL,dt)
implicit none
type(mesh_matrix),intent(in) :: mesh
real(dp),intent(in) :: CFL
real(dp),intent(out) :: dt
real(dp),intent(in),dimension(:,:) :: x,y
! local variables
real(dp),dimension(1:mesh%nel) :: nl12,nl13,nl23,sper,area
! Calculate the time scale from the CFL number
! computation of the dtscale number as function of the inscribed circle
!$omp parallel
!$omp workshare
nl12=sqrt((x(:,1)-x(:,2))**2.0_dp+(y(:,1)-y(:,2))**2.0_dp)
nl13=sqrt((x(:,1)-x(:,3))**2.0_dp+(y(:,1)-y(:,3))**2.0_dp)
nl23=sqrt((x(:,2)-x(:,3))**2.0_dp+(y(:,2)-y(:,3))**2.0_dp)
sper=(nl12+nl13+nl23)/2.0_dp
area=sqrt(sper*(sper-nl12)*(sper-nl13)*(sper-nl23))
!$omp end workshare
!$omp end parallel
dt = minval(area/sper)*1.0_dp/3.0_dp*2.0_dp/3.0_dp*CFL
end subroutine dtscale

subroutine DrmatrixT3(r,s,x,y,jac,invjac,phi,dphidx,dphidy)
implicit none
! Drmatrix evaluates derivatives of triangular shape functions
! In the notation of NUDG, Hestehaven et. al.
! input
! r    reference x coordinate
! s    reference y coordinate
! x    physical x vertex coordinates
! y    physical y vertex coordinates
! output
! jac   The jacobian
! invjac The inverse of jacobian
! phi   The shape functions
! dphidx The x derivatives of phi
! dphidy The y derivatives of phi
real(dp), intent(in), dimension(:) :: x,y
real(dp), intent(in) :: r,s
real(dp), intent(out), dimension(:) :: dphidx,dphidy,phi
real(dp), intent(out):: jac,invjac
! Local variables
integer :: nel,nnd,iv
real(dp),dimension(1:size(x,1)) :: dphidr,dphids,phi_g
real(dp) :: dxdr,dxds,dydr,dyds
nnd=size(x,1)

! Get the shape functions, e.g. the geometric interpolation functions.
call T3shape(r,s,phi_g,dphidr,dphids)

dxdr = 0.0_dp; dxds = 0.0_dp; dydr = 0.0_dp
dyds = 0.0_dp; jac = 0.0_dp; invjac = 0.0_dp
!
do iv = 1,nnd
   dxdr = dxdr + x(iv)*dphidr(iv)
dxds = dxds + x(iv)*dphids(iv)
   dydr = dydr + y(iv)*dphidr(iv)
   dyds = dyds + y(iv)*dphids(iv)
end do
jac = dxdr*dyds - dxds*dydr
! check The shape of the elements if the Area is less than 1e-9 then there is
! probably something wrong
if (jac < 1e-9) then
   print *, 'Bad element ...'
   if (jac <= 0.0) then
      print *, 'error - Aborted ...'
      stop
   end if
end if
invjac = 1.0_dp/jac
!
do iv = 1,nnd
  phi(iv) = phi_g(iv)
  dphidx(iv) = dphidr(iv)*dyds - dphids(iv)*dydr
  dphidy(iv) = -dphidr(iv)*dxds + dphids(iv)*dxdr
end do
end subroutine DrmatrixT3

subroutine assembleConT3(par,mesh,x,y,flowxx,flowyy,viscosity,triplet, C, K, A, f)
  implicit none
  type(par_matrix),intent(in) :: par
  type(mesh_matrix),intent(in) :: mesh
  type(COO_matrix),intent(in) :: triplet
  real(dp),intent(in),dimension(:,:,1:2) :: x,y
  real(dp),intent(in),dimension(:,:,1:2) :: viscosity,flowxx,flowyy
  real(dp),intent(out),dimension(1:triplet%nnz) :: C,K,A
  real(dp),intent(inout),dimension(:,:,1:2) :: f

  ! Local Variables
  integer :: nnd,nng,krow,kcol,igp,iv,i,j,ii,jj,nnz,ival,iel,th
  real(dp),dimension(1:mesh%nel,1:mesh%nnd,1:mesh%nnd) :: ke,ae,ce,tke,tae,tce
  real(dp),dimension(1:mesh%nel,1:mesh%nnd) :: fe,tfe
  real(dp),dimension(1:mesh%nel) :: rhs,flowx,flowy
  real(dp),dimension(1:mesh%nnd) :: phi,dphidx,dphidy
  real(dp),dimension(1:mesh%nnd,2) :: Gp
  real(dp),dimension(1:mesh%nnd) :: Gw
  real(dp) :: jac,invjac,rp,sp,t1,t2

  nnd=size(mesh%EToV,2)
  ! Defining the element coordinate vector and shape functions
  !$omp parallel
  !$omp workshare
  K = 0.0_dp; A = 0.0_dp; C = 0.0_dp; rhs=0.0_dp
  ke = 0.0_dp; ae = 0.0_dp; ce = 0.0_dp; fe = 0.0_dp;
  tke=0.0_dp;tae=0.0_dp;tce=0.0_dp;tfe=0.0_dp
  flowx=flow(:,:,1); flowy=flow(:,:,2)
  !$omp end workshare nowait
  !$omp end parallel
  ! setting up the Gaussian Quadrature points
  call gaussGQ(3,Gw,Gp)
  nng=size(Gp,1)
  ! inner loop over elements
  !loop over Gauss points in this case 3 times.
  do igp = 1,nng
    rp=Gp(igp,1)
    sp=Gp(igp,2)
call windrot(rp, sp, x, y, flowx, flowy)
rhs = cone(rp, sp, x, y)

!$omp parallel private(iel, i, j, jac, invjac, phi, dphidx, dphidy) default(shared)
!$omp do !schedule(dynamic)
    ! evaluate derivatives and interpolate the local polynomial function
    ! over the triangle shape element.
    do iel=1, mesh%nel
        call DrmatrixT3(rp, sp, x(iel,:), y(iel,:), jac, invjac, phi, dphidx, dphidy)
        do i = 1, nnd
            do j = 1, nnd
                tke(iel, i, j) = viscosity(1)*dphidx(i)*dphidx(j)*invjac + &
                    viscosity(2)*dphidy(i)*dphidy(j)*invjac
                tce(iel, i, j) = phi(i)*phi(j)*jac
                tae(iel, i, j) = flowx(iel)*phi(i)*dphidx(j) + &
                    flowy(iel)*phi(i)*dphidy(j)
            end do
        tfe(iel, i) = rhs(iel)*phi(i)*jac
    end do

!$omp end do nowait
!$omp workshare
    ke=ke+tke
    ae=ae+tae
    ce=ce+tce
    fe=fe+tfe
!$omp end workshare nowait
!$omp end parallel
end do

!$omp parallel private(th, iel, krow, ii, kcol, jj, ival, i) default(shared)
!$omp do
    do th=1, par%npar
        do i=par%startblock(th,1), par%endblock(th,1)
            iel=par%matrix(i,1); krow=par%matrix(i,2)
            kcol=par%matrix(i,3); ival=par%matrix(i,4)
            K(ival) = K(ival) + ke(iel, krow, kcol)
            C(ival) = C(ival) + ce(iel, krow, kcol)
            A(ival) = A(ival) + ae(iel, krow, kcol)
        end do
    do i=par%startblock(th,2), par%endblock(th,2)
        iel=par%vector(i,1); krow=par%vector(i,2)
        ii=par%vector(i,3)
        f(ii) = f(ii) + fe(iel, krow)
    end do
end do
!$omp end do nowait
!$omp end parallel
end subroutine assembleConT3

subroutine peclet(x,y,flowx,flowy, viscosity, epe, eph, epw)
imPLICIT NONE
real(dp),intent(in),dimension(:,:) :: x,y
real(dp),intent(in),dimension(:) :: flowx,flowy,viscosity
real(dp),intent(out),dimension(:) :: epe,eph,epw
! local variables
real(dp),dimension(1:size(x,1)) :: nl12,nl13,nl23,flow_l2,sper,area,flow_h
integer :: i
! Calculate the Peclet number
! computation of element Peclet number (at the centroid)
! rectangle specific calculation here
!$omp parallel
!$omp workshare
nl12=sqrt((x(:,1)-x(:,2))**2.0_dp+(y(:,1)-y(:,2))**2.0_dp)
nl13=sqrt((x(:,1)-x(:,3))**2.0_dp+(y(:,1)-y(:,3))**2.0_dp)
nl23=sqrt((x(:,2)-x(:,3))**2.0_dp+(y(:,2)-y(:,3))**2.0_dp)
sper=(nl12+nl13+nl23)/2.0_dp
flow_l2 = sqrt(flowx(:) * flowx(:) + flowy(:) * flowy(:))
area=sqrt(sper*(sper-nl12)*(sper-nl13)*(sper-nl23))
!$omp end workshare nowait
!$omp end parallel
if(all(flowx==0.0_dp)) then
flow_h=nl23
else if( all(flowy==0.0_dp)) then
flow_h=nl13
else
!$omp parallel default(__AUTO)
!!$omp do
! do i=1,size(x,1)
! flow_h(i) = max(nl12(i), nl23(i), nl13(i))
! end do
!!$omp workshare
!$omp end do workshare nowait
!$omp end parallel
end if
!$omp parallel
!$omp workshare
eph = flow_h
epe = flow_h*flow_12/2.0_dp
epw = flow_12
!$omp end workshare nowait
!$omp end parallel
if(all(viscosity==0.0_dp)) then
  !$omp parallel
  !$omp workshare
epe=0.5_dp
  !$omp end workshare nowait
  !$omp end parallel
else
  !$omp parallel
  !$omp workshare
epe = epe/sum(viscosity*0.5_dp)
  !$omp end workshare nowait
  !$omp end parallel
end if
epe=abs(0.5_dp*(1.0_dp-1.0_dp/epe))
!print *, 'maximum element Peclet number is',maxval(epe)
end subroutine peclet

subroutine Adv_SUPGT3(par,mesh,x,y, flowxx,flowyy,viscosity,triplet, A, f)
  implicit none
type(par_matrix),intent(in) :: par
  type(mesh_matrix),intent(in) :: mesh
type(COO_matrix),intent(in) :: triplet
real(dp),intent(in),dimension(:,:) :: x,y
real(dp),intent(in),dimension(:) :: viscosity,flowxx,flowyy
real(dp),intent(out),dimension(:) :: A
real(dp),intent(inout),dimension(:) :: f

! Local Variables
integer :: nnd,nng,kcol,igp,iv,i,j,ii,jj,nnz,ival,iel,th
real(dp),dimension(1:mesh%nel,1:3,1:3) :: ae,tae
real(dp),dimension(1:mesh%nel,1:3) :: fe,tfe
real(dp),dimension(1:mesh%nel) :: rhs,epe,eph,epw,lpe,flowx,flowy
real(dp),dimension(1:3) :: phi,dphidx,dphidy
real(dp),dimension(1:3,2) :: Gp
real(dp),dimension(1:3) :: Gw
real(dp) :: rp,sp,jac,invjac,onethree=1.0_dp/3.0_dp,t1,t2
nnd=size(mesh%EToV,2)
! Setting the rhs vector and flow vectors
 ! setting up the Gaussian Quadrature points
 call gaussGQ(3,Gw,Gp)
 nng=size(Gp,1)
 ! inner loop over elements
 ! loop over Gauss points in this case 3 times.
 !t1=omp_get_utime()
 do igp = 1,nng
   rp=Gp(igp,1)
   sp=Gp(igp,2)
   ! evaluate derivatives and interpolate the local polynomia function
   ! over the triangle shape element.
   call windrot(rp,sp,x,y,flowx,flowy)
   rhs = cone(rp,sp,x,y)
 !$omp parallel private(iel,i,j,jac,invjac,phi,dphidx,dphidy) default(shared)
 !$omp do
   do iel=1,mesh%nel
     call DrmatrixT3(rp,sp,x(iel,:),y(iel,:),jac,invjac,phi,dphidx,dphidy)
     do i = 1,nnd
       do j = 1,nnd
         tae(iel,i,j) = flowx(iel)*dphidx(i)*flowx(iel)*dphidx(j)*invjac + 
                         flowy(iel)*dphidy(i)*flowx(iel)*dphidx(j)*invjac + 
                         flowx(iel)*dphidx(i)*flowy(iel)*dphidy(j)*invjac + 
                         flowy(iel)*dphidy(i)*flowy(iel)*dphidy(j)*invjac
       end do
       tfe(iel,i) = rhs(iel)* phi(i)*jac
     end do
   end do
 !$omp end do nowait
 !$omp workshare
 ae=ae+tae
 fe=fe+tfe
 !$omp end workshare
 !$omp end parallel
 end do
 !t2=omp_get_utime();print *,’SUPG Dr Matrix: ’,t2-t1
 ! Scale with the Peclet number N = N_a+ h/2*U_i/U^2*dN_a/dx
 call windrot(onethree,onethree,x,y,flowx,flowy)
call peclet(x, y, flowx, flowy, viscosity, epe, eph, epw)
lpe=epe*(eph/epw)
!t1=omp_get_wtime()
!$omp parallel private(th, iel, krow, kcol, jj, ival, i) default(shared)!
reduction(+: K, C, A)
!$omp do
   do th=1,par%npar
     !
     print *,par%startblock(th, 2)
     !
     print *,par%endblock(th, 2)
   do i=par%startblock(th, 1),par%endblock(th, 1)
     iel=par%matrix(i, 1); krow=par%matrix(i, 2)
     kcol=par%matrix(i, 3); ival=par%matrix(i, 4)
     A(ival) = A(ival) + lpe(iel)*ae(iel, krow, kcol)
   end do
   do i=par%startblock(th, 2),par%endblock(th, 2)
     iel=par%vector(i, 1); krow=par%vector(i, 2)
     ii=par%vector(i, 3)
     f(ii) = f(ii) + lpe(iel)*fe(iel, krow)
   end do
end do
!$omp end do nowait
!$omp end parallel
!t2=omp_get_wtime(); print *, 'SUPG Reduction: ', t2-t1
end subroutine Adv_SUPGT3

subroutine T3shape(r, s, phi, dphidr, dphids)

implicit none
real(dp), intent(in) :: r, s
real(dp), intent(out), dimension(:) :: phi, dphidr, dphids
!
shape evaluates triangular shape functions
!
in the notation of NUDG, Hestehaven et. al.
!
input
!
r x coordinate
!
s y coordinate
!
output
!
phi shape function
!
dphidr x derivative of phi
!
dphids y derivative of phi
!
phi(1) =1.0_dp-r-s
phi(2) = r
phi(3) = s
dphidr(1) = -1.0_dp
dphidr(2) = 1.0_dp
dphidr(3) = 0.0_dp  
dphids(1) = -1.0_dp  
dphids(2) = 0.0_dp  
dphids(3) = 1.0_dp  
end subroutine T3shape

subroutine windrot(r,s,xl,yl,flowx,flowy)
  implicit none
  real(dp),intent(in),dimension(:,:) :: xl,yl
  real(dp),intent(in) :: r,s
  real(dp),intent(inout),dimension(:) :: flowx,flowy

  ! local variables
  integer :: iv,nnd
  real(dp) :: xc,yc
  real(dp),dimension(1:size(xl,1)) :: x,y
  real(dp),dimension(1:size(xl,2)) :: dphids,dphidt,phi

  ! interpolate the wind fields at the gaussian quadratic points
  ! Find the the reference coordiantes in the reference element
  ! and build the mapping from the reference element to the physical element.
  !
  ! r   reference x coordinate
  ! s   reference y coordinate
  ! xl  physical x vertex coordinates
  ! yl  physical y vertex coordinates
  nnd=size(xl,2);x=0.0_dp;y=0.0_dp
  call T3shape(r,s,phi,dphids,dphidt)
  do iv=1,nnd
    !$omp parallel
    !$omp workshare
    x = x + phi(iv)*xl(:,iv)
    y = y + phi(iv)*yl(:,iv)
    !$omp end workshare nowait
    !$omp end parallel
  end do

  ! make a rotational wind with center in main mesh
  !$omp parallel
  !$omp workshare
  xc=(maxval(x)+minval(x))*0.5_dp
  yc=(maxval(y)+minval(y))*0.5_dp
  flowx=-(y-yc)
  flowy=(x-xc)
  !$omp end workshare nowait
  !$omp end parallel
end subroutine windrot

function cone(r,s,xl,yl)
    implicit none
    real(dp),intent(in) :: r,s
    real(dp),intent(in),dimension(:,::) :: xl,yl
    real(dp),dimension(1:size(xl,1)) :: cone
    ! local variables
    integer :: iv,nns,nnd
    real(dp) :: xc,yc,h,rc,xlt,ylt
    real(dp),dimension(1:size(xl,1)) :: x,y,rr
    real(dp),dimension(1:size(xl,2)) :: dphids,dphidt,phi

    ! interpolate the cone at the gaussian quadratic points
    ! Find the the reference coordiantes in the reference element
    ! and build the mapping from reference element to the phyiscal element.
    ! r reference x coordinate
    ! s reference y coordinate
    ! xl physical x vertex coordinates
    ! yl physical y vertex coordinates
    nnd=size(xl,2);nns=size(xl,1);x=0.0_dp;y=0.0_dp
    call T3shape(r,s,phi,dphids,dphidt)
    do iv=1,nnd
        !$omp parallel
        !$omp workshare
        x = x + phi(iv)*xl(:,iv)
        y = y + phi(iv)*yl(:,iv)
        !$omp end workshare
        !$omp end parallel
    end do
    ! make a Cone with center at xc,yc
    !$omp parallel private(iv) default(shared)
    !$omp workshare
    xlt=maxval(x)+minval(x)
    ylt=maxval(y)+minval(y)
    xc=xlt*0.25_dp
    yc=ylt*0.5_dp
    h=10000.0_dp
    rc=0.125_dp*xlt
    cone=0.0_dp
    rr=sqrt((x-xc)**2.0_dp + (y-yc)**2.0_dp)
    !$omp end workshare
    !$omp do
do iv=1,nns
   if (rr(iv) < rc) then
      cone(iv)=h*(1.0-rr(iv)/rc)
   else
      cone(iv)=0.0_dp
   end if
end do
!$omp end do nowait
!$omp end parallel
end function cone

subroutine gaussGQ(quad_num,quad_w,quad_xy)
  implicit none
  integer :: quad_num
  ! Based on the subroutine quad_rule by John Burkardt
  ! https://people.scs.fsu.edu/~burkardt/f_src/f_src.html
  real (dp) :: a,b,c,d,e,f,g,h,t,u,v,w
  real (dp), dimension(quad_num) :: quad_w
  real (dp), dimension(2,quad_num) :: quad_xy
  if ( quad_num == 1 ) then
    quad_xy(1:2,1:quad_num) = reshape ( (/ &
      1.0_dp / 3.0_dp, 1.0_dp / 3.0_dp /), (/ 2, quad_num /) )
    quad_w(1:quad_num) = 1.0_dp
  else if ( quad_num == 3 ) then
    quad_xy(1:2,1:quad_num) = reshape ( (/ &
      0.5_dp, 0.0_dp, &
      0.5_dp, 0.5_dp, &
      0.0_dp, 0.5_dp /), (/ 2, quad_num /) )
    quad_w(1:quad_num) = 1.0_dp / 3.0_dp
  else if ( quad_num == 4 ) then
    a=1.0_dp/sqrt(3.0_dp)
    quad_xy(1:2,1:quad_num) = reshape ( (/ &
      -a, -a, &
      a, -a, &
      a, a, &
      -a, a /), (/ 2, quad_num /) )
  end if
end subroutine gaussGQ
quad_w(1:quad_num) = (/ 1.0_dp, 1.0_dp, 1.0_dp, 1.0_dp /)

else if ( quad_num == 6 ) then

  a = 0.816847572980459_dp
  b = 0.091576213509771_dp
  c = 0.108103018168070_dp
  d = 0.445948490915965_dp
  v = 0.109951743655322_dp
  w = 0.223381589678011_dp

quad_xy(1:2,1:quad_num) = reshape ( (/ &
  a, b, &
  b, a, &
  b, b, &
  c, d, &
  d, c, &
  d, d /), (/ 2, quad_num /) )

quad_w(1:quad_num) = (/ v, v, v, w, w /)

else if ( quad_num == 9 ) then

  a = 0.124949503233232_dp
  b = 0.43752524838384_dp
  c = 0.797112651860071_dp
  d = 0.165409927389841_dp
  e = 0.037477420750088_dp
  u = 0.205950504760887_dp
  v = 0.063691414286223_dp

quad_xy(1:2,1:quad_num) = reshape ( (/ &
  a, b, &
  b, a, &
  b, b, &
  c, d, &
  c, e, &
  d, c, &
  d, e, &
  e, c, &
  e, d /), (/ 2, quad_num /) )

quad_w(1:quad_num) = (/ u, u, u, v, v, v, v /)
else

    write (*, '(a)')
    write (*, '(a)') 'QUAD_RULE - Fatal error!'
    write (*, '(a,i8)') ' No rule is available of order QUAD_NUM = ', &
    quad_num
    stop
end if
end subroutine gaussGQ

function vtkoutput(mesh,un,t) result(ier)
use LIB_VTK_IO
implicit none
type(mesh_matrix),intent(in) :: mesh
integer,intent(in) :: t
integer :: ier
real(dp),intent(in),dimension(:) :: un

! Local Variables
integer,dimension(1:mesh%nel*3) :: connect,nctype
character :: time*4,filename*20
write(time,'(I4.4)') t
filename='AdvDiff'//trim(time)//'.vtk'
connect=reshape(mesh%EToV,(/3*mesh%nel/))-1
nctype=5
ier=VTK_INI('BINARY',trim(filename),'Mesh','UNSTRUCTURED_GRID')
ier=VTK_GEO(mesh%nns,mesh%vx,mesh%vy,mesh%vx*0.0_dp)
ier=VTK_CON(mesh%nel,connect,nctype)
ier=VTK_DAT(mesh%nns,'node')
ier=VTK_VAR(mesh%nns,'cone',un)
! ier=VTK_VAR('vect',nvtx,'velocity',flowx,flowy,flowy*0.0)
ier=VTK_END()
end function vtkoutput
end module precondition

A.4 Solver

MODULE solver
use precondition
use sparse_precondition
use utils, only : print_matrix
use omp_lib
use superlumt
use sunsolver
A THE CODE

contains

subroutine driver(par,mesh,triplet,viscosity,CFL,theta,time)
  implicit none
  type(mesh_matrix), intent(in) :: mesh
  type(par_matrix),intent(in) :: par
  type(COO_matrix),intent(inout) :: triplet
  real(dp),intent(in) :: CFL,theta,time
  real(dp),intent(inout),dimension(:) :: viscosity
  ! local variables
  integer :: info,t,ipivot(1:mesh%nns),ier=0,msglvl=0,i,steps
  integer,dimension(1:mesh%nel) :: offset,NCtype
  real(dp),dimension(1:mesh%nel,1:3) :: x,y ! Containing the element numbers.
  real(dp),dimension(1:triplet%nnz) :: SS,RR,A,K,C,Asupg,flowx,flowy
  real(dp),dimension(1:mesh%nns) :: un,e,b,f,fsupg
  real(dp) :: dt1,dt2,handle(150),t1,t2,t3,dt
  type(CSC_matrix) :: S,R
  t3=omp_get_wtime()
  allocate(R%values(1:triplet%nnz),S%values(1:triplet%nnz))
  R%nnz=triplet%nnz;S%nnz=triplet%nnz
  flowx=0.0_dp;flowy=0.0_dp
  call omp_set_num_threads(par%npar)
  do i = 1,3
    !$omp parallel
    !$omp workshare
    x(:,i) = mesh%vx(mesh%EToV(:,i))
    y(:,i) = mesh%vy(mesh%EToV(:,i))
    !$omp end workshare nowait
    !$omp end parallel
  end do
  call dtscale(mesh,x,y,CFL,dt)
  steps=floor(time/dt)
  !print *,"Time step is: ",dt
  ! Assemble the Arrays
  t1=omp_get_wtime()
  call assembleConT3(par,mesh,x,y,flowx,flowy,viscosity,triplet,C, K, A, f)
  t2=omp_get_wtime();print *,"assembleConT3 time: ",t2-t1
  !$omp parallel
  !$omp workshare
  A=A+K
  !$omp end workshare nowait
  !$omp end parallel
  ! If needed call the SUPG diffusion matrix
  t1=omp_get_wtime()
  call Adv_SUPGT3(par,mesh,x,y,flowx,flowy,viscosity,triplet, Asupg, fsupg)
t2=omp_get_wtime();print *,'Adv_SUPGT3 time: ',t2-t1
!$omp parallel
!$omp workshare
A=A+Asupg
f=f+fsupg
!$omp end workshare nowait
!$omp end parallel
!print *,'Assemble okay'
! Step 3, Update C -> S and A -> R
call dirichletBC(mesh, triplet, A,C,f)
!print *,'Dirichlet okay'
dt1=theta*dt
dt2=(1.0_dp-theta)*dt
!$omp parallel
!$omp workshare
SS = C - dt2*A  ! S is now S, R is still A, formula (4.35) % Step 3
RR = SS + dt*A  ! R is now R, formula (4.36) % Step 3
!$omp end workshare nowait
!$omp end parallel
! Step 4, Prepare solving
! Coverting the sparse storage triplet COO format to the sparse storage
! format that is supported by The Super LU solver, i.e. CSC sparse storage format
! (The Super LU is a C written solver, so rember to take care of the 0 based
! array format for C vs. the 1 based array storage format.)
call coo_to_csc0(triplet%row,triplet%col, RR, R)
!print *,'triplet to csc - zero based: okay'
! Converting the Mass matrix from triplet COO storage format to csc storage format.
! Here we don’t need C array storage format therefore we call the conversion routine
! with iflag=1. The Mass matrix has to be multiply with the solution vector in the time
! loop. Where we use the routine csc1_vector_mult(Ap,Ai,A,x,b) from thesparse_utils module.
! We could also have used the sparsekit routine amux, however, then we had to define to new
! row and column vectors. Here we just use the csc type structure.
call coo_to_csc1(triplet%row,triplet%col, SS, S)
!print *,'triplet to csc - one based: okay'
un=f
!ier=vtkoutput(mesh,un,1)
!write(0100,*)un
t2=omp_get_wtime()
print *,'Assemble time: ',t2-t3
do t = 2,50
   t1 = omp_get_wtime()
   ! Step 5
call csc1_vector_mult(S%colptr,S%rowind,S%values,un,e)
   un=e
! Solve Step 10
! One-call routine of SPSOLVE
$omp parallel
$omp single
call solverMT(par%npar,R,un,1,mesh%nns)
$omp end single
$omp end parallel
!call solversun(R,un)
t2 = omp_get_wtime()
print *, 'Solver times: ', t2-t1
!ier=vtkoutput(mesh,un,t)
!write(0100,*)un
end do
close(010)
end subroutine driver
END MODULE solver

A.5 SuperLu interface

module superlumt
  use precision
  use omp_lib
  contains
  subroutine solverMT(nprocs,csc,U,nrhs,ldb)
    implicit none
    type(CSC_MATRIX),intent(in) :: csc
    real(dp),intent(inout),dimension(:) :: U
    integer,intent(in) :: nrhs,ldb,nprocs
    ! local variables
    integer :: n,nnz,info,nprocs
    n=size(csc%colptr)-1
    nnz=csc%nnz
    call c_bridge_pdgssv(nprocs, n, nnz, nrhs,csc%values, &
      csc%rowind,csc%colptr,U,ldb,info)
    if(info/=0) print *, 'Solution error'
  end subroutine solverMT
end module superlumt

A.6 Precision

MODULE Precision
  INTEGER,PARAMETER:: dp=SELECTED_REAL_KIND(15,307)
  INTEGER,PARAMETER:: qp=SELECTED_REAL_KIND(33,4931)
INTEGER,PARAMETER:: i4=SELECTED_INT_KIND(4)
INTEGER,PARAMETER:: i8=SELECTED_INT_KIND(8)
INTEGER,PARAMETER:: i10=SELECTED_INT_KIND(10)

type CSC_matrix
  real(dp), dimension(:,), pointer :: values => NULL()
  integer, dimension(:,), pointer :: rowind => NULL()
  integer, dimension(:,), pointer :: colptr => NULL()
  integer :: status,nnz,n
end type CSC_matrix
type COO_matrix
  real(dp), dimension(:,), pointer :: values => NULL()
  integer, dimension(:,), pointer :: row => NULL()
  integer, dimension(:,), pointer :: col => NULL()
  integer :: status,nnz
end type COO_matrix
type mesh_matrix
  real(dp), dimension(:,), pointer :: vx => NULL()
  real(dp), dimension(:,), pointer :: vy => NULL()
  integer, dimension(:,,:), pointer :: EToV => NULL()
  integer, dimension(:,,:), pointer :: EToE => NULL()
  integer, dimension(:,,:), pointer :: EToF => NULL()
  integer, dimension(:,), pointer :: BCTYPE => NULL()
  logical, dimension(:,), pointer :: BC => NULL()
  integer :: status,nns,nel,nne,nnd
end type mesh_matrix
type par_matrix
  integer, dimension(:,,:), pointer :: matrix => NULL()
  integer, dimension(:,,:), pointer :: vector => NULL()
  integer, dimension(:,,:), pointer :: startblock => NULL()
  integer, dimension(:,,:), pointer :: endblock => NULL()
  integer :: npar
end type par_matrix
END MODULE Precision