Parallel Discontinuous Galerkin Method

Yin Ki, Ng

The Chinese University of Hong Kong

7 August, 2015

1 Abstract

Discontinuous Galerkin Method(DG-FEM) is a class of Finite Element Method (FEM) for finding approximation solutions to systems of differential equations that can be used to simulate real life problems such as chemical transport phenomena.

In contrast to the standard FEM, DG-FEM allows discontinuity of the cell boundary solutions (a cell refers to a finite sub-domain from a defined partition of the domain) (c.f. [1]) which makes parallelization on this method flavourable.

This project is going to explore how we may construct the parallel code for 1D DG-FEM that can be scaled in existing supercomputers. The Poisson's equation is to be used as a demonstration.

The mathematics behind the method is introduced in **Section 2**, where one can look into the weak formulation with the bilinear function in the 1D case (**Section 2.1**, **2.2**) and in the 2D case (**Section 2.3**). An 1D step-by-step example is given in **Section 3**.

The design of parallelization in the 1D case can be found in **Section 4**. The motivation is explained in **Section 4.1** , and the parallel code structure, which takes the construction of the linear system and the solver of the linear system into consideration, would be discussed in **Section 4.2** and **4.3** respectively.

Finally, future work is discussed in **Section 5** and more result of the project work can be found in the **Appendix**.

2 Understanding DG-FEM

Overview: *In this section , we are going to understand the mathematics behind DG-FEM with a start-off at FEM. "Jump condition" is the major component that makes DG-FEM stand out from FEM.*

2.1 1-Dimensional DG-FEM

Consider the following 1D-problem, involving a differential equation with boundary conditions:

$$\begin{cases} -u'' = f \\ u(a) = u(b) = 0 \end{cases} \tag{1}$$

We wish to solve the unknown **trial** function u on the domain $I = [a, b]$. We may do so by multiplying an arbitrary **test** function v defined such that it has the same boundary condition $v(a) = v(b) = 0$ as u. We integrate over the domain to get

$$-\int (u'' + f)v = -\int u''v - \int fv = 0 \tag{2}$$

Finite Element Method (FEM) Suppose we choose the test function v to be continuous and differentiable everywhere on the domain. Integration by part on the first term gives

$$\int u'v' - u'v|_a^b - \int fv = 0$$

The central term disappears so we have

$$\int u'v' - \int fv = 0 \tag{3}$$

(Notice that instead of having a twice derivative term in equation (2) (**strong form**), now we get a first derivative term in equation (3) (**weak form**). Cut the domain I into intervals $I_0 = [x_0, x_1], I_1 = [x_1, x_2], \cdots I_N = [x_N, x_{N+1}]$, we then try to approximate u by choosing our basis function (or **shape function**) $\theta = [\theta_1 \dots \theta_N]$

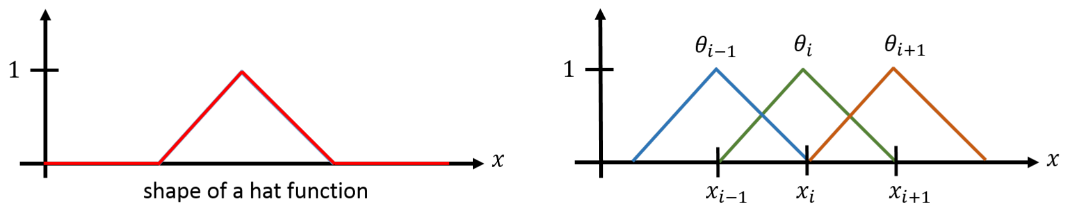


Figure 2.1.a: Example of basis functions: Hat functions

that would satisfy **partition of unity**, and approximate u as a linear combination of θ:

$$u_h = \sum_{i=1}^{i=N} \alpha_i \theta_i \in V_h \tag{4}$$

where $V_h = span(\theta)$. Moreover, we may also make the choice $v_h \in V_h$, that is, make u_h and v_h both as a linear combination of θ. We choose $v_h = \theta\bar{v}$, where $\bar{v} = [\bar{v}_1 ... \bar{v}_N]^T$ are completely arbitrary values at the discrete points. Substituting and rewriting equation (3) we get

$$\int u_h' v_h' - \int f\, v_h = \bar{v}^T \Big(\int (\theta^T)'(\theta)'\alpha - \int \theta^T f \Big) = 0$$

which must hold for all arbitrary values of $\bar{v}$, so

$$\underbrace{\int (\theta^T)'(\theta)'}_{S}\ \alpha - \underbrace{\int \theta^T f}_{r} = 0 \tag{5}$$

where the **stiffness matrix** S is positive definite. Since we approximate the solution by partitioning the domain into finite elements (intervals), this is called the **Finite Element Method (FEM)** and we are left with solving the standard problem $S\alpha = r$.

Discontinuous Galerkin Method (DG-FEM) Suppose now we want our choice of the test function v being discontinuous on the nodes $x_1, \cdots, x_N$. Then we cannot integrate the first term $-\int u''v$ in equation (2) since v is not differentiable on the whole domain.

On each node x_j, let x_j^- represent the end of the left interval I_{j-1}, and x_j^+ represent the start of the right interval I_j. Notice that we would have our function v look like this:

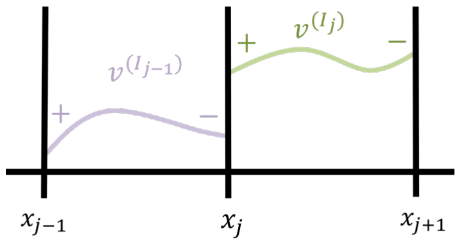


Figure 2.1.b: v being discontinuous on nodes

thus we are having two function v values, $v(x_j^-)$ from I_{j-1} and $v(x_j^+)$ from I_j, on each node x_j for $j = 1...N$. We may proceed by choosing a new basis function $\phi = [\phi_1 \dots \phi_M]$ that is discontinuous at the nodes

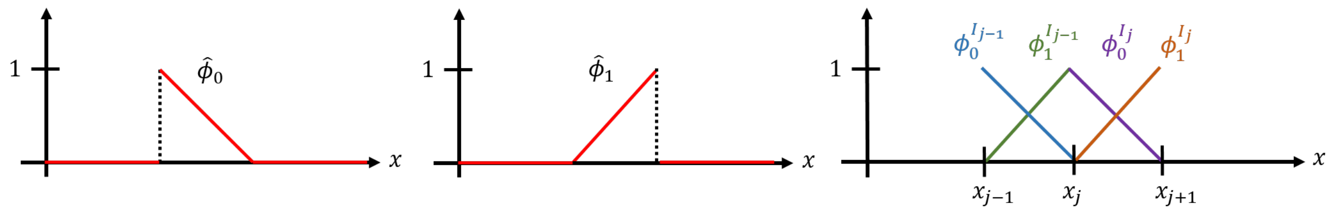


Figure 2.1.c: Example of linear basis functions, discontinuous at nodes

and make our choice of the test function v to be ϕ. So now we can do integration by part on each interval $[x_j, x_{j+1}]$ and get:

$$
\begin{aligned}
-\int u''v &= -\sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u''v \\
&= \sum_{j=0}^{j=N} \left(\int_{x_j}^{x_{j+1}} u'v' + u'(x_j^+)\, v(x_j^+) - u'(x_{j+1}^-)\, v(x_{j+1}^-) \right) \\
&= \sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u'v' + u'(x_0^+)\, v(x_0^+) + \Bigg(\left(-u'(x_1^-)\, v(x_1^-) + u'(x_1^+)\, v(x_1^+) \right) + \ldots \\
&\quad + \left(-u'(x_N^-)\, v(x_N^-) + u'(x_N^+)\, v(x_N^+) \right) \Bigg) - u'(x_{N+1}^-)\, v(x_{N+1}^-) \\
&= \sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u'v' + u'(x_0^+)\, v(x_0^+) + \sum_{j=1}^{j=N} \boldsymbol{[u'v]_{x_j}} - u'(x_{N+1}^-)\, v(x_{N+1}^-) \qquad (6)
\end{aligned}
$$

of which $\boldsymbol{[u'v]_{x_j}} = -u'(x_j^-)\, v(x_j^-) + u'(x_j^+)\, v(x_j^+)$ are called the **jumps**. Now we may define $u'(x_0^-)\, v(x_0^-) = u'(x_{N+1}^+)\, v(x_{N+1}^+) = 0$ and after adding them to the equation, we have

$$
-\int u''v = \sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u'v' + \sum_{j=0}^{j=N+1} [u'v]_{x_j} \qquad (7)
$$

In order to transmit information between each jump, we define $\boldsymbol{\{\cdot\}_j}$, $\boldsymbol{[\cdot]_j}$ be the average and difference of the function values on x_j respectively, and rewrite each jump as

$$
\begin{aligned}
[u'v]_{x_j} &= -u'(x_j^-)\, v(x_j^-) + u'(x_j^+)\, v(x_j^+) \\
&= \{\frac{1}{2}\left(u'(x_j^+) + u'(x_j^-) \right)\}\, [v(x_j^+) - v(x_j^-)] + [u'(x_j^+) - u'(x_j^-)]\, \{\frac{1}{2}\left(v(x_j^+) + v(x_j^-) \right)\} \\
&= \{u'\}_j [v]_j + [u']_j \{v\}_j \qquad (8)
\end{aligned}
$$

The second term $[u']_j\{v\}_j$ disappears since u is continuous. On the other hand, we wish to make the equation symmetric on u, v, so we add symmetric terms $\{v'\}_j[u]_j$ for each j. Together with the **penalty term**, we obtain the **weak form** with $a(u, v)$ which is a bilinear function symmetric on u, v:

$$
a(u,v) \equiv \sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u'v' + \sum_{j=0}^{j=N+1} \left(\{u'\}_j [v]_j + \underbrace{\{v'\}_j [u]_j}_{\text{symmetric term}} \right) + \underbrace{\gamma \sum_{j=0}^{N+1} \frac{1}{|I_j|} [u]_j [v]_j}_{\text{penalty term}} \qquad (9)
$$

(Notice that when the test function v is continuous, the weak form is essentially the same as the one in FEM [1]).

[1]since the second term and the penalty term would disappear.

Assume u is to be approximated by a linear combination of ϕ

$$u_h = \sum_{j=1}^{j=M} \alpha_j \phi_j \tag{10}$$

so we have

$$\begin{aligned}
a(u_h, v_h) &= \int f\, v_h + \text{symmetric term} + \text{penalty term} \\
a\Big(\sum_{j=1}^{j=M} \alpha_j \phi_j, \phi_i\Big) &= \int f\, \phi_i + \text{symmetric term} + \text{penalty term} \qquad \text{for } i = 1...M \\
\sum_{j=1}^{j=M} \underbrace{a(\phi_j, \phi_i)}_{S_{ij}} \alpha_j &= \underbrace{\int f\, \phi_i}_{r_i} + \text{symmetric term} + \text{penalty term} \qquad \text{for } i = 1...M \qquad (11)
\end{aligned}$$

Again we are left with the standard problem $S\alpha = r$. This is called the **Discontinuous Galerkin Finite Element Method (DG-FEM)**.

2.2 Essence of the Penalty Term

Suppose we have

$$\sum_{I_j} \int_{x_j}^{x_{j+1}} u'v' + \sum_{I_j} \Big(\{u'\}_j [v]_j + \{v'\}_j [u]_j \Big) = \int fv$$

and we wish to solve for u. Writing in matrix form, it is easy to see that the stiffness matrix in LHS may be singular [2]. Therefore we wish to add a **penalty term** to LHS such that the matrix is always invertible and the solution can be uniquely determined. Therefore we define

$$a(u,v) \equiv \sum_{I_j} \int_{x_j}^{x_{j+1}} u'v' + \sum_{I_j} \Big(\{u'\}_j [v]_j + \{v'\}_j [u]_j \Big) + \underbrace{\gamma \sum_{I_j} \frac{1}{|I_j|} [u]_j [v]_j}_{\text{penalty term}} \tag{12}$$

and we claim that the stiffness matrix S defined by $a(u,v)$ is positive definite. We are going to show that, for a chosen A, if γ is large enough we would always have

$$a(u,v) \geq A\, ||u||_{1,h}{}^2 \quad \forall u \in V_h \tag{13}$$

where V_h is the finite element(DG) space and $||u||_{1,h}$ is a well-defined norm. Then we know that S is positive definite and thus invertible, and we are done.

With the conventional norm $||\cdot||_I = (\cdot,\cdot)_I^{\frac{1}{2}}$ in which the inner product is defined as $(f,g)_I = \int_I fg$ for any functions f, g, we can now define $||u||_{1,h}$:

[2] Take u to be a non-zero constant function and LHS would become zero.

Definition 2.2.1:

$$||u||_{1,h} = \Big(\sum_{I_j} ||u'||_{I_j}{}^2 + \sum_{I_j} |\, \{u'\,\}_j|^2\, |I_j| + \sum_{I_j} \frac{\gamma}{|I_j|} |\, [u]\, |_j{}^2 \Big)^{\frac{1}{2}}$$

and $||u||_{1,h}$ is a well-defined norm. In particular, it satisfies the norm property

$$||u||_{1,h} \geq 0 \quad and \quad ||u||_{1,h} = 0 \iff u = 0$$

Proof: " $\Leftarrow$ " is trivial.
" $\Rightarrow$ " : We must have both $||u'||_{I_j}{}^2$ and $|\,\{u'\}_j\,|^2$ equal to zero. The former term equals to zero implies that u is a piecewisely constant function . The latter term equals to zero implies that there are no differences between the function values of u on all nodes including the boundary, whose value is defined to be zero. Hence we must have u being a constant zero function.

Before proceeding we would need the following:

***Theorem 2.2.2*:** (*Trace Inequality*) Let v be a function on domain Ω and H be the length of Ω.

Then we have

$$\int_{\partial\Omega} |v|^2 ds \;\leq\; cH^{-1}\, ||v||_{\Omega}{}^2 + cH\, ||\nabla||_{\Omega}{}^2$$

for some constant c. Or in 1D sense,

$$|v(a)|^2 + |v(b)|^2 \;\leq\; cH^{-1}\, ||v||_I{}^2 + cH\, ||v'||_I{}^2 \tag{14}$$

with $H = |I|$ being the length of the interval.

***Theorem 2.2.3*:** (*Inverse Inequality*) Let u be a function on the interval I. Then we have

$$||u'||^2 \;\leq\; c\, |I|^{-2} ||u||_I{}^2$$

Or

$$|I|^2 ||u'||^2 \;\leq\; c\, ||u||_I{}^2 \tag{15}$$

for some constant c.

***Claim 2.2.4*:** We have

$$\sum_{I_j} |I_j|\, |\{u'\}_j|^2 \;\leq\; c \sum_{I_j} ||u'||_{I_j}{}^2$$

for some constant c.

Proof: Applying *Trace Inequality* and *Inverse Inequality*, we have

$$\begin{aligned}
& \sum_{I_j} |I_j|\,|\{u'\}_j|^2 \\
(\textit{Trace.}, \text{ equation } (14)) \leq \quad & \sum_{I_j} |I_j| \Big(\hat{c}\,|I_j{}^{-1}| ||u'||_{I_j}{}^2 + \hat{c}\,|I_j|\,||u''||_{I_j}{}^2 \Big) \\
= \quad & \hat{c} \sum_{I_j} \Big(|I_j||I_j|^{-1} ||u'||_{I_j}{}^2 + |I_j|^2\,||u''||_{I_j}{}^2 \Big) \\
= \quad & \hat{c} \sum_{I_j} \Big(||u'||_{I_j}{}^2 + |I_j|^2\,||u''||_{I_j}{}^2 \Big) \\
(\textit{Inverse.}, \text{ equation } (15)) \leq \quad & \hat{c} \sum_{I_j} \Big(||u'||_{I_j}{}^2 + \check{c}\,||u'||_{I_j}{}^2 \Big) \\
= \quad & c \sum_{I_j} ||u'||_{I_j}{}^2
\end{aligned}$$

for some constant c.

***Claim 2.2.5*:** We have

$$\sum_{I_j} \{u'\}_j\,[u]_j \;\leq\; c_1\,\delta \sum_{I_j} ||u'||_{I_j}{}^2 + \frac{c_2}{\delta} \sum_{I_j} \frac{1}{I_j} |\,[u]_j\,|^2$$

for some constant c_1, c_2.

Proof:

$$\begin{aligned}
& \sum_{I_j} \{u'\}_j\,[u]_j \\
\leq \quad & \sum_{I_j} |\{u'\}_j||[u]_j| \\
(\textit{AM-GM.}) \leq \quad & \sum_{I_j} \Big(\frac{\delta}{2} |I_j||\{u'\}_j|^2 + \frac{1}{\delta|I_j|} [u]_j{}^2 \Big) \\
(\textit{Claim 2.2.4}) \leq \quad & \sum_{I_j} c_1\,\delta\,||u'||_{I_j}{}^2 + \sum_{I_j} \frac{1}{\delta|I_j|} [u]_j{}^2 \\
= \quad & c_1\,\delta \sum_{I_j} ||u'||_{I_j}{}^2 + \frac{c_2}{\delta} \sum \frac{1}{|I_j|} |u|_{I_j}{}^2
\end{aligned}$$

for arbitrary constant δ and some constant c_1, c_2.

Finally cleaning up:

$$
\begin{aligned}
& a(u,u) - A\,||u||_{1,h}{}^2 \\
\text{(equation (12), } \textit{Def. 2.2.1}) = \quad & \Big(\sum_{I_j} ||u'||_{I_j}{}^2 - 2\sum_{I_j}\{u'\}_j[u']_j + \sum_{I_j}\frac{\gamma}{|I_j|}|[u]_j|^2\Big) \\
& - A\Big(\sum_{I_j} ||u'||_{I_j}{}^2 + \sum_{I_j}|I_j||\{u'\}_j|^2 + \sum_{I_j}\frac{\gamma}{|I_j|}|[u]_j|^2\Big) \\
= \quad & (1-A)\sum_{I_j}||u'||_{I_j}{}^2 + (1-A)\sum_{I_j}\frac{\gamma}{I_j}|[u]_j|^2 \\
& - 2\sum_{I_j}\{u'\}_j[u']_j - A\sum_{I_j}|I_j||\{u'\}_j|^2 \\
(\textit{Claim 2.2.4, 2.2.5}) \geq \quad & (1-A)\sum_{I_j}||u'||_{I_j}{}^2 + (1-A)\sum_{I_j}\frac{\gamma}{I_j}|[u]_j|^2 \\
& - \Big(c_1\,\delta\sum_{I_j}||u'||_{I_j}{}^2 + \frac{c_2}{\delta}\sum\frac{1}{|I_j|}|u|_{I_j}{}^2\Big) - c_3 A\sum_{I_j}||u'||_{I_j}{}^2 \\
= \quad & (1 - A - c_1\delta - c_3 A)\sum_{I_j}||u'||_{I_j}{}^2 + (\gamma - \gamma A - \frac{c_2}{\delta})\sum_{I_j}\frac{1}{|I_j|}|[u]_j|^2 \\
\geq \quad & 0
\end{aligned}
$$

for γ large enough[3] according to the chosen A and δ, which is equivalent to equation (13). *Q.E.D.*

[3]For instant, choose $A = 1/(2c_3)$ and $\delta = -1/(2c_1c_3)$, then take $\gamma \geq (4c_1c_2c_3{}^2)/(1-2c_3)$.

2.3 2-Dimensional DG-FEM

Now we may consider the general problem:

$$\begin{cases} -\triangle u &= f & \text{in } \Omega \\ u &= g_D & \text{on } \Gamma_D \\ \frac{\partial u}{\partial \mathbf{n}} &= g_N & \text{on } \Gamma_N \end{cases} \tag{16}$$

which is the **Poisson's Equation** $-\triangle u = f$, where f is given and u is the unknown function that describes the temperature on a domain Ω, together with the given temperature g_D on boundary Γ_D (**Dirichlet boundary condition**) and the given **heat flux**[4] g_N on boundary Γ_N (**Neumann boundary condition**).

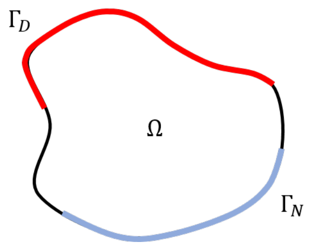


Figure 2.3.a: Example of a 2D domain

Recall that in 1D problem with domain $I = [a, b]$, solving by FEM we multiply both sides of the equation by an arbitrary continuous function v and do integration by part

$$-\int_a^b u''v\,dx = \int_a^b u'v'\,dx - u'v|_a^b = \int_a^b fv\,dx$$

Notice that, by Fundamental Theorem of Calculus, the term $u'v|_a^b$ is indeed

$$u'v|_a^b = (u'v)(b) - (u'v)(a) = (u'v)(b)\cdot\mathbf{n}_b + (u'v)(a)\cdot\mathbf{n}_a$$

where $\mathbf{n}_b = 1$, $\mathbf{n}_a = -1$ are the norm derivatives at boundary nodes. Similarly, in the general problem we have:

$$\begin{aligned} -\int_\Omega \triangle u\, v\,dx &= \int_\Omega \nabla u\cdot\nabla v\,dx - \int_{\partial\Omega} (\nabla u\cdot\mathbf{n})\,v\,ds \\ &= \int_\Omega \nabla u\cdot\nabla v\,dx - \int_{\partial\Omega} \frac{\partial u}{\partial \mathbf{n}} v\,ds \\ &= \int_\Omega fv\,dx \end{aligned} \tag{17}$$

in which we have reduced the equation from the strong form to a more desirable weak form.

Now we take a look at the 2D problem. We would like to approximate u by a linear combination of some chosen basis functions in a finite element space. We may consider partitioning the domain into finite elements in the form of triangles:

[4]The rate of heat energy transfer through a surface.

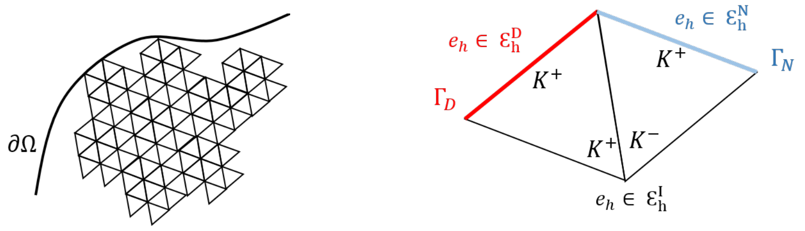

$\partial\Omega$
$e_h \in \mathcal{E}_h^D$
$e_h \in \mathcal{E}_h^N$
K^+
K^+
Γ_N
Γ_D
K^+
K^-
$e_h \in \mathcal{E}_h^I$

Figure 2.3.b: Partition of a 2D domain in triangles and edge sets: $\mathcal{E}_h^I$, $\mathcal{E}_h^D$ and $\mathcal{E}_h^N$

Denote a triangle by K and its diameter by h, we let $\mathcal{T}_h$ be the set of all triangles and define the following sets:

$$\begin{cases} \mathcal{E}_h^I & = \{e_h \,:\, e_h = \partial K^+ \cap \partial K^- \quad \forall K \in \mathcal{T}_h\}\text{: set of } \textbf{interior edges} \\ \mathcal{E}_h^D & = \{e_h \,:\, e_h = \partial K^+ \cap \ \Gamma_D \quad \forall K \in \mathcal{T}_h\}\text{: set of } \textbf{boundary edges on } \Gamma_D \\ \mathcal{E}_h^N & = \{e_h \,:\, e_h = \partial K^+ \cap \ \Gamma_N \quad \forall K \in \mathcal{T}_h\}\text{: set of } \textbf{boundary edges on } \Gamma_N \\ \mathcal{E}_h & = \mathcal{E}_h^I \cup \mathcal{E}_h^D. \end{cases} \tag{18}$$

(Note that the signs of the triangles are declared relatively; and interior edges must align with triangles of different signs. Triangles at the boundary are always denoted by K^+.) Thus now we can make our choice of basis functions to be discontinuous across the edges, and proceed on DG-FEM. Take v to be the chosen basis functions, we then have

$$\begin{aligned} -\int_\Omega \triangle u\, v\, dx &= -\sum_{K\in\mathcal{T}_h}\int_K \triangle u\, v\, dx \\ &= \sum_{K\in\mathcal{T}_h}\int_K \nabla u\cdot\nabla v\, dx - \sum_{K\in\mathcal{T}_h}\int_{\partial K}\frac{\partial u}{\partial \mathbf{n}} v\, ds \\ &= \sum_{K\in\mathcal{T}_h}\int_K \nabla u\cdot\nabla v\, dx - \sum_{e_h\in\mathcal{E}_h^D}\int_{e_h}\frac{\partial u}{\partial \mathbf{n}} v\, ds - \sum_{e_h\in\mathcal{E}_h^N}\int_{e_h}\frac{\partial u}{\partial \mathbf{n}} v\, ds \\ &\qquad - \sum_{e_h\in\mathcal{E}_h^I}\int_{e_h}\Big(\frac{\partial u^+}{\partial \mathbf{n}n^+}v^+ + \frac{\partial u^-}{\partial \mathbf{n}n^-}v^-\Big)ds \\ &= \int_\Omega f v\, dx \end{aligned} \tag{19}$$

We wish to find out a symmetric bilinear function $a_h(u,v)$, whose stiffness matrix is positive definite, to solve for the unknown u. For simplicity, we denote

$$\int_k fg\, dx \;\equiv\; (f,g)_K \qquad \text{and} \qquad \int_{e_h} fg\, ds \;\equiv < f,g >_{e_h}$$

Also we define the notations $\{\cdot\}$, $[\cdot]$ on function and norm derivative:

$$\begin{aligned} \{u\} &\equiv \frac{1}{2}(u^+ + u^-) \qquad &\text{and}\qquad [u] &\equiv u^+ - u^- \\ \{\partial_n u\} &\equiv \frac{1}{2}\Big(\frac{\partial u^+}{\partial \mathbf{n}^+} + \frac{\partial u^-}{\partial \mathbf{n}^+}\Big) \qquad &\text{and}\qquad [\partial_n u] &\equiv \frac{\partial u^+}{\partial \mathbf{n}^+} - \frac{\partial u^-}{\partial \mathbf{n}^+} \end{aligned} \tag{20}$$

Then look into the term regarding the set of interior edges. By noticing the direction of the norm derivatives and the continuity of u, we have

$$\begin{aligned}
\sum_{e_h \in \mathcal{E}_h^I} \int_{e_h} \left(\frac{\partial u^+}{\partial \mathbf{n}^+} v^+ + \frac{\partial u^-}{\partial \mathbf{n}^-} v^- \right) ds &= \sum_{e_h \in \mathcal{E}_h^I} \int_{e_h} \left(\frac{\partial u^+}{\partial \mathbf{n}^+} v^+ - \frac{\partial u^-}{\partial \mathbf{n}^+} v^- \right) ds \\
&= \sum_{e_h \in \mathcal{E}_h^I} \Big(< \frac{\partial u^+}{\partial \mathbf{n}^+}, v^+ >_{e_h} - < \frac{\partial u^-}{\partial \mathbf{n}^+}, v^- >_{e_h} \Big) \\
\text{(after rearranging)} \;&=\; \sum_{e_h \in \mathcal{E}_h^I} \Big(< \{\partial_n u\}, [v] >_{e_h} + < [\partial_n u], \{v\} >_{e_h} \Big) \\
\text{(since } [\partial_n u] = 0) &= \sum_{e_h \in \mathcal{E}_h^I} < \{\partial_n u\}, [v] >_{e_h} \qquad (21)
\end{aligned}$$

Rewrite equation (19) with the notations defined, we have :

$$\begin{aligned}
-(\triangle u, v) \quad &= \quad \sum_{K \in \mathcal{T}_h} (\nabla u, \nabla v)_K - \sum_{e_h \in \mathcal{E}_h^I} < \{\partial_n u\}, [v] >_{e_h} - \sum_{e_h \in \mathcal{E}_h^D} < \partial_n u, v >_{e_h} \\
&\qquad - \sum_{e_h \in \mathcal{E}_h^N} < \partial_n u, v >_{e_h} \\
&= \quad (f, v) \qquad (22)
\end{aligned}$$

To make the expression in the left hand side symmetric on u, v, we add the artificial terms

$$- \sum_{e_h \in \mathcal{E}_h^I} < \{\partial_n v\}, [u] >_e \qquad \text{and} \qquad - \sum_{e_h \in \mathcal{E}_h^D} < \partial_n v, u >_{e_h}$$

Notice that the first term equals to 0 since $[u] = 0$ by continuity of u. In the second term, $u = g_D$ which is given, so we add this term to both sides of the equation. Also $\sum_{e_h \in \mathcal{E}_h^N} < \partial_n u, v >_{e_h} = \sum_{e_h \in \mathcal{E}_h^N} < g_N, v >$ is known, so this term is moved to the right hand side.

To make the bilinear form coercive (positive definite), we add the penalty terms

$$\sum_{e_h \in \mathcal{E}_h^I} \frac{\gamma}{|e_h|} < [u], [v] >_{e_h} = 0 \quad \text{and} \quad \sum_{e_h \in \mathcal{E}_h^D} \frac{\gamma}{|e_h|} < u, v >_{e_h} = \sum_{e_h \in \mathcal{E}_h^D} \frac{\gamma}{|e_h|} < g_D, v >_{e_h} .$$

The first and second terms are added to the left hand side, while the third term is added to the right hand side. With the unknown values stay in the left hand side and away from the right hand side, we may define the left hand side as a bilinear form $a_h(u, v)$:

$$\begin{aligned}
a_h(u, v) \equiv \quad & \sum_{K \in \mathcal{T}_h} (\nabla u, \nabla v)_K - \sum_{e_h \in \mathcal{E}_h^I} \Big(< \{\partial_n u\}, [v] >_{e_h} + < \{\partial_n v\}, [u] >_{e_h} - \frac{\gamma}{|e_h|} < [u], [v] >_{e_h} \Big) \\
& - \sum_{e_h \in \mathcal{E}_h^D} \Big(< \partial_n u, v >_{e_h} + < \partial_n v, u >_{e_h} - \frac{\gamma}{|e_h|} < u, v >_{e_h} \Big) \qquad (23)
\end{aligned}$$

Note that the penalty terms are added to the interior and Dirichlet edges but not to the Neumann edges. This is because the Neumann boundary terms have been moved to the right hand side.

Also, adopting the notation that on the boundary edges

$$\{\partial_n u\} = \partial_n u, \quad \{\partial_n v\} = \partial_n v, \quad [u] = u, \quad [v] = v,$$

then we can write the bilinear form in equation (25) as a more compact form (recall that $\mathcal{E}_h = \mathcal{E}_h^I \cup \mathcal{E}_h^D$)

$$a_h(u,v) \equiv \sum_{K\in\mathcal{T}_h} (\nabla u, \nabla v)_K - \sum_{e_h\in\mathcal{E}_h} \Big(< \{\partial_n u\}, [v] >_{e_h} + < \{\partial_n v\}, [u] >_{e_h} - \frac{\gamma}{|e_h|} < [u], [v] >_{e_h} \Big) \tag{24}$$

We also define the right hand side, which depends only on v and is composited of known values, as a function $F(v)$ by

$$F(v) \equiv (f,v) + \sum_{e_h\in\mathcal{E}_h^D} \Big(\frac{\gamma}{|e_h|} < g_D, v >_{e_h} - < g_D, \partial_n v >_{e_h} \Big) + \sum_{e_h\in\mathcal{E}_h^N} < g_N, v >_{e_h} \tag{25}$$

Thus, we have shown the following: If u satisfies the problem (16) and belongs to $C^1(\Omega)$ then u satisfies the equation (26) for all $v \in E_h$ (E_h is the space of **test functions**).

IN GENERAL. The **DG finite dimensional space** is defined by

$$V_h = \{v_h \mid v_h|_K \in \mathcal{P}_q, \ \forall K \in \mathcal{T}_h\}$$

where $\mathcal{P}_q$ is the vector space of polynomials of total degree q in the variables x, y in two dimensions and x, y, z in three dimensions.

We now can define the **DG approximation** $u_h \in V_h$ of u as follows: Find $u_h \in V_h$ satisfying

$$a_h(u_h, v_h) = F(v_h), \quad \forall v_h \in V_h. \tag{26}$$

The existence and uniqueness of u_h are consequences of the following result:

Introducing the norm $\|\cdot\|_{1,h} : E_h \to R$ defined by

$$\|v\|_{1,h} = \Big\{ \sum_{K\in\mathcal{T}_h} \|\nabla v\|_K^2 + \sum_{e_h\in\mathcal{E}_h} |e_h| \, |\{\partial_n v\}|_{e_h}^2 + \sum_{e_h\in\mathcal{E}_h} \frac{\gamma_h}{|e_h|} |[v]|_{e_h}^2 \Big\}^{1/2},$$

we have

Lemma 2.3 For the bilinear form $a_h(\cdot,\cdot)$ the following continuity and coercivity properties hold

$$|a_h^{\gamma_h}(u,v)| \le \|u\|_{1,h} \|v\|_{1,h}, \quad \forall u, v \in E_h.$$

There exist positive constants γ and c_a depending only on q and the shape regularity of the cells in $\mathcal{T}_h$ such that if $\gamma_h \ge \gamma$, then

$$a_h^{\gamma_h}(v,v) \ge c_a \|v\|_{1,h}^2, \quad \forall v \in V^h, \tag{27}$$

3 An 1D example: $u = x(2 - x)$

Overview: *In this section, we are going to see an example of solving the heat equation using DG-FEM. The main idea is to carefully construct the corresponding stiffness matrix from the bilinear function defined and then solve for the linear system.*

Recall the 1D problem (1) in **Section 2.1**. Suppose now the problem is defined on $I = [0, 1]$ and we have $f = -2$, $u(0) = 0$ and $u(1) = 1$. The solution can be found explicitly as $u = x(2 - x)$, but now we try to solve for the approximation u_h using DG-FEM.

Step 1. Partition the domain The accuracy of the approximation as well as the computational cost increase with the number of cells (which are intervals in 1D). In this example, we take number of cells equal to 4 (so $N = 3$): $I_0 = [0, 0.25]$, $I_1 = [0.25, 0.5]$, $I_2 = [0.5, 0.75]$ and $I_3 = [0.75, 1]$.

Step 2. Choose the basis functions We want each $u_h^{(I_j)}$ to be approximated by a linear combination of a set of basis functions within each interval I_j. These chosen basis functions also take the role as the test function v. We may make our choice of the basis functions to be linear, quadratic or polynomials in higher degree to improve the performance.

Legendre polynomials and lagrange polynomials are some of the popular choices. In this example, the set of linear lagrange polynomials $\hat{\phi}_0 = 1 - x$, $\hat{\phi}_1 = x$ over the **master interval** $[0, 1]$ is chosen for use. It would need to be mapped into the corresponding set of functions $\phi_0^{(I_j)}, \phi_1^{(I_j)}$ over the **local interval** I_j for each j afterwards to serve our purpose.

Step 3. Construct the matrix In equation (11) we know that each entry S_{ij} of our stiffness matrix S is defined by the bilinear symmetric function $a(\phi_j, \phi_i)$. In our choice of ϕ, each of the four intervals has 2 degree of freedom. Thus S is a 8×8 matrix, with each parameter of a runs across all $\phi_0^{(I_0)}, \phi_1^{(I_0)}, \phi_0^{(I_1)}, \phi_1^{(I_1)}, \ldots, \phi_0^{(I_3)}, \phi_1^{(I_3)}$.

$$\left[\begin{array}{cc|ccc}
a(\phi_0^{(I_0)}, \phi_0^{(I_0)}) & a(\phi_1^{(I_0)}, \phi_0^{(I_0)}) & a(\phi_0^{(I_1)}, \phi_0^{(I_0)}) & a(\phi_1^{(I_1)}, \phi_0^{(I_0)}) & \cdots \\
a(\phi_0^{(I_0)}, \phi_1^{(I_0)}) & a(\phi_1^{(I_0)}, \phi_1^{(I_0)}) & a(\phi_0^{(I_1)}, \phi_1^{(I_0)}) & a(\phi_1^{(I_1)}, \phi_1^{(I_0)}) & \\
 & & & & \\
a(\phi_0^{(I_1)}, \phi_0^{(I_1)}) & a(\phi_1^{(I_1)}, \phi_0^{(I_1)}) & \ddots & & \\
a(\phi_0^{(I_1)}, \phi_1^{(I_1}) & a(\phi_1^{(I_1)}, \phi_1^{(I_1)}) & & & \\
\vdots & & & &
\end{array}\right]$$

Note that this **global** matrix is composed by blocks of 2×2 **local** matrices, and each block stores information of basis functions interaction within or across intervals. The four diagonal blocks store those interactions **within an interval**, the six sub-diagonal blocks store those interactions **across adjacent intervals**, and other blocks store zeros – since the basis function values are non-zero only if they are within or on the boundary of their defined intervals.

Recall that a is defined as

$$a(u,v) \equiv \sum_{j=0}^{j=N} \int_{x_j}^{x_{j+1}} u'v' + \sum_{j=0}^{j=N+1} \Big(\{u'\}_j[v]_j + \underbrace{\{v'\}_j[u]_j}_{\text{symmetric term}} \Big) + \underbrace{\gamma \sum_{j=0}^{N+1} \frac{1}{|I_j|}[u]_j[v]_j}_{\text{penalty term}} \tag{9}$$

where $\{\cdot\}_j$ is the average of jump on x_j and $[\cdot]_j$ is the jump on x_j as defined in equation (8). Let $k_u, k_v \in \{0,1\}$ be the indices of basis functions within the set that serve the role for the trial function u and the test function v respectively. We now look into the computation of each part of the equation.

Step 3.1. Compute $\int_{I_j} u'v'$. From the summation sign in equation (9), we know that this term is computed with respect to intervals, so only the diagonal blocks are involved. The term

$$\int_{I_j} \phi_{k_u}^{\prime(I_j)} \phi_{k_v}^{\prime(I_j)}$$

may be computed by first considering $\phi_{k_u}^{\prime(I_j)} \phi_{k_v}^{\prime(I_j)}$. It can be mapped from $\hat{\phi}'_{k_u} \hat{\phi}'_{k_v}$ with a note to the change in slope of the basis functions from the master interval $[0,1]$ to the local interval I_j.

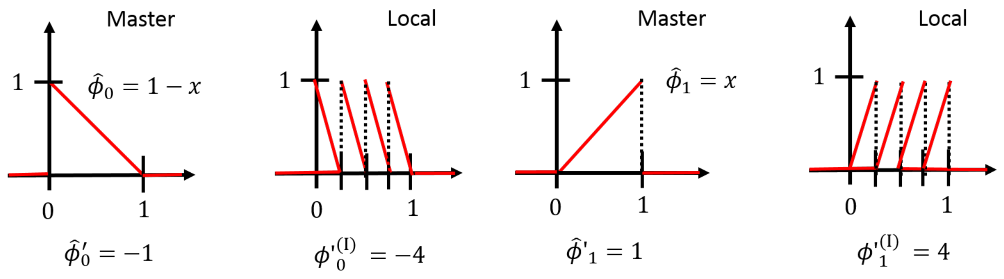


Figure 3.a: Mapping from master interval to local interval

Afterwards integration can be done explicitly or using Gaussian Quadratures.

Step 3.2. Compute $\{u'\}_j[v]_j + \{v'\}_j[u]_j$. This term is computed with respect to nodes which stand between adjacent intervals, so for each node x_j both the $(j-1)$-th, j-th diagonal block and the related sub-diagonal blocks are involved. Now since

$$\{\phi'_{k_u}\}_j[\phi_{k_v}]_j + \{\phi'_{k_v}\}_j[\phi_{k_u}]_j$$

is symmetric on ϕ_{k_u}, ϕ_{k_v}, so we may first look at the left term. After expansion we get the expression

$$\begin{aligned} &\{\phi'_{k_u}\}_j[\phi_{k_v}]_j \\ &= \frac{1}{2}\Big(\phi'_{k_u}(x_j^+) + \phi'_{k_u}(x_j^-)\Big)\Big(\phi_{k_v}(x_j^+) - \phi_{k_v}(x_j^-)\Big) \\ &= \frac{1}{2}\phi'_{k_u}(x_j^+)\phi_{k_v}(x_j^+) - \frac{1}{2}\phi'_{k_u}(x_j^+)\phi_{k_v}(x_j^-) + \frac{1}{2}\phi'_{k_u}(x_j^-)\phi_{k_v}(x_j^+) - \frac{1}{2}\phi'_{k_u}(x_j^-)\phi_{k_v}(x_j^-) \end{aligned} \tag{28}$$

which contains four terms: the "++" term, "+−" term, "−+" and "−−" term respectively.

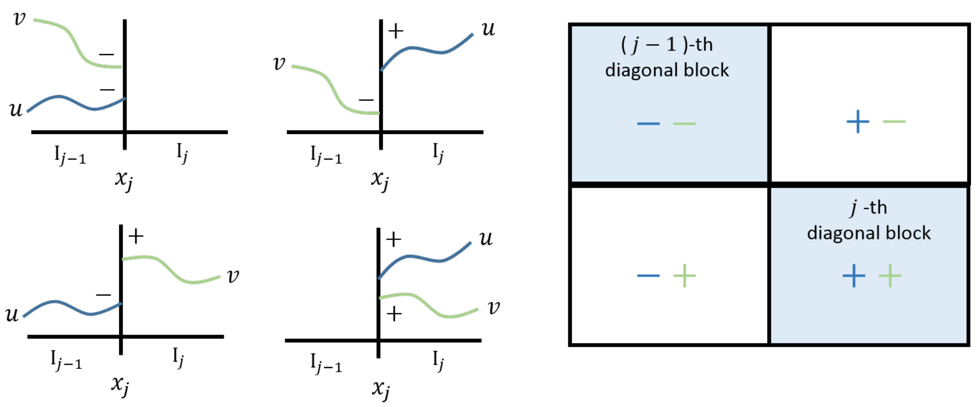


Figure 3.b: the "++" term, "+−" term, "−+" term and "−−" term and their positions

Recall in Figure 2.1.b that x_j^+ and x_j^- denote the right interval and left interval from x_j respectively, thus we may now see that the "++" term is non-zero only if both ϕ'_{k_u} and ϕ_{k_v} are from I_j ; so this term is added to the j-th diagonal block. The "+−" term is non-zero only if ϕ_{k_v} is on I_{j-1} and ϕ'_{k_u} is on I_j; so this term is added to the right sub-diagonal block of the $(j-1)$-th diagonal block. Similarly the "−+" term is added to the left sub-diagonal block of the j-th diagonal block, and the "−−" term is added to the $(j-1)$-th diagonal block.

The expansion of the right term $\{\phi'_{k_u}\}_j[\phi_{k_v}]_j$ is similar and the resultant four terms may be added to the corresponding blocks accordingly with the same principle.

Step 3.3. Compute the penalty term. This term is computed with respect to nodes. Again we may expand and get the four "++", "+−", "−+", "−−" terms:

$$\begin{aligned}&\frac{\gamma}{|I_j|}[\phi_{k_u}]_j[\phi_{k_v}]_j\\&=\frac{\gamma}{|I_j|}\big(\phi_{k_u}(x_j^+)-\phi_{k_u}(x_j^-))\big)\big(\phi_{k_v}(x_j^+)-\phi_{k_v}(x_j^-)\big)\\&=\frac{\gamma}{|I_j|}\phi_{k_u}(x_j^+)\phi_{k_v}(x_j^+)-\frac{\gamma}{|I_j|}\phi_{k_u}(x_j^+)\phi_{k_v}(x_j^-)-\frac{\gamma}{|I_j|}\phi_{k_u}(x_j^-)\phi_{k_v}(x_j^+)+\frac{\gamma}{|I_j|}\phi_{k_u}(x_j^-)\phi_{k_v}(x_j^-)\end{aligned}$$

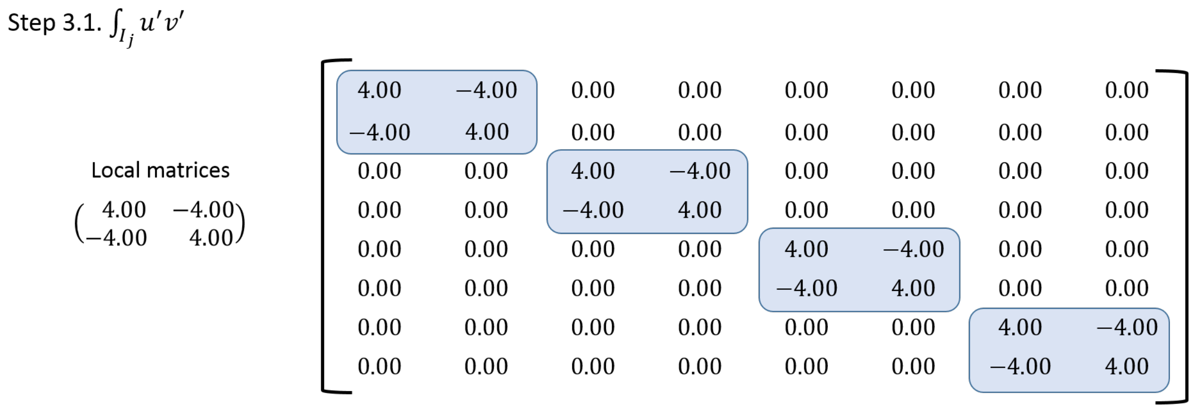
and added to the corresponding blocks respectively following **Step 3.2**. (note that when ϕ_{k_u}, ϕ_{k_v} come from adjacent intervals, the value $|I_j|$ in the term is computed as the average of $|I_{j-1}|$ and $|I_j|$ instead). The choice of γ is arbitrary – as long as it is "large enough" (see **Section 2.2**) with respect to the degree of polynomial chosen for the basis functions. In this example, we choose $\gamma = 5$.

Note 1: At boundary nodes. For **Step 3.2** and **3.3**, we have to be more careful when working on the boundary nodes. Clearly at x_0, we have only the " $++$" term, while at x_{N+1} we have only the " $--$" term, since on these two nodes we are working on intervals I_0 and I_N only and their adjacent intervals are not defined .

For the same reason, an "average jump" $\{\cdot\}$ of a function on the boundary node is defined to be the function value at that node itself within the defined interval (while a "jump" $[\cdot]$ is defined as the function value of x_j^+ minus by that of x_j^- for all nodes. Recall that on boundary nodes, we define $x_0^- = x_{N+1}^+ = 0$). For example, at **Step 3.2**, we have the term $\{\phi'_{k_u}\}_0[\phi_{k_v}]_0 = \phi'_{k_u}(x_0^+)\phi_{k_v}(x_0^+)$ and $\{\phi'_{k_u}\}_{N+1}[\phi_{k_v}]_{N+1} = \phi'_{k_u}(x_{N+1}^-)\big(-\phi_{k_v}(x_{N+1}^-)\big)$, which are added to the 0-th diagonal block and N-th diagonal block respectively.

Note 2: Pre-computation. Before **Step 3**, we may pre-compute the terms $\hat{\phi}_{k_u}\hat{\phi}_{k_v}$, $\hat{\phi}'_{k_u}\hat{\phi}_{k_v}$, $\hat{\phi}'_{k_u}\hat{\phi}'_{k_v}$ for each $k_u, k_v = \{0, 1\}$ and for each "$++$", "$+-$", "$-+$", "$--$" term on the master interval. They will come into handy when we are working on the bilinear function – since mapping the computed terms from the master interval to local intervals requires only easy linear transformation.

The below shows the actual figures from each step of the matrix construction in our example:

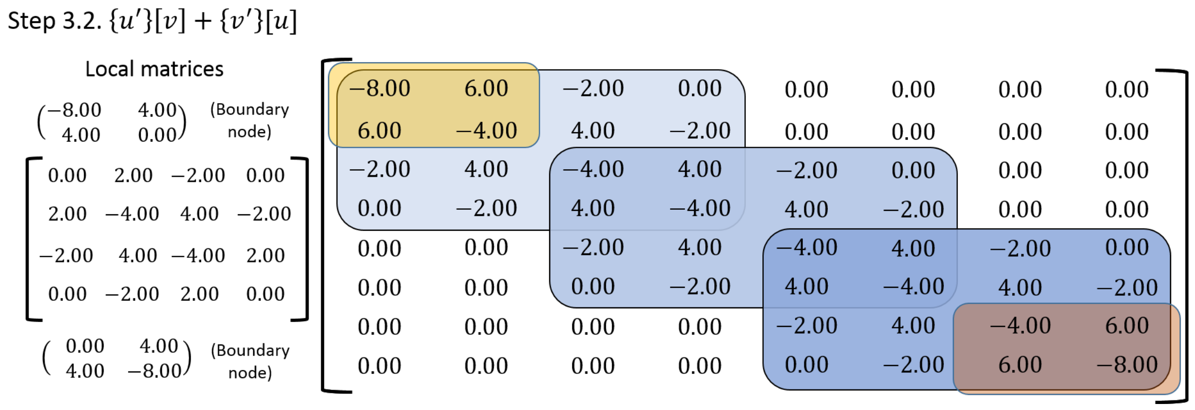


Step 3.1. $\int_{I_j} u'v'$

Local matrices

$$\begin{pmatrix} 4.00 & -4.00 \\ -4.00 & 4.00 \end{pmatrix}$$

$$\begin{bmatrix}
4.00 & -4.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
-4.00 & 4.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 4.00 & -4.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & -4.00 & 4.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 4.00 & -4.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & -4.00 & 4.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 4.00 & -4.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -4.00 & 4.00
\end{bmatrix}$$



Step 3.2. $\{u'\}[v] + \{v'\}[u]$

Local matrices

$$\begin{pmatrix} -8.00 & 4.00 \\ 4.00 & 0.00 \end{pmatrix}$$ (Boundary node)

$$\begin{bmatrix}
0.00 & 2.00 & -2.00 & 0.00 \\
2.00 & -4.00 & 4.00 & -2.00 \\
-2.00 & 4.00 & -4.00 & 2.00 \\
0.00 & -2.00 & 2.00 & 0.00
\end{bmatrix}$$

$$\begin{pmatrix} 0.00 & 4.00 \\ 4.00 & -8.00 \end{pmatrix}$$ (Boundary node)

$$\begin{bmatrix}
-8.00 & 6.00 & -2.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
6.00 & -4.00 & 4.00 & -2.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
-2.00 & 4.00 & -4.00 & 4.00 & -2.00 & 0.00 & 0.00 & 0.00 \\
0.00 & -2.00 & 4.00 & -4.00 & 4.00 & -2.00 & 0.00 & 0.00 \\
0.00 & 0.00 & -2.00 & 4.00 & -4.00 & 4.00 & -2.00 & 0.00 \\
0.00 & 0.00 & 0.00 & -2.00 & 4.00 & -4.00 & 4.00 & -2.00 \\
0.00 & 0.00 & 0.00 & 0.00 & -2.00 & 4.00 & -4.00 & 6.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -2.00 & 6.00 & -8.00
\end{bmatrix}$$

Step 3.3. $\frac{\gamma}{|I_j|}[u][v]$

Local matrices

$\begin{pmatrix} 20.00 & 0.00 \\ 0.00 & 0.00 \end{pmatrix}$ (Boundary node)

$$\begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 20.00 & -20.00 & 0.00 \\ 0.00 & -20.00 & 20.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix}$$

$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 20.00 \end{pmatrix}$ (Boundary node)

$$\begin{bmatrix} 20.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 20.00 & -20.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & -20.00 & 20.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 20.00 & -20.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & -20.00 & 20.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 20.00 & -20.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -20.00 & 20.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 20.00 \end{bmatrix}$$

Finish. $a(u, v)$

Final matrix

$$\begin{bmatrix} 16.00 & 2.00 & -2.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 2.00 & 20.00 & -16.00 & -2.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ -2.00 & -16.00 & 20.00 & 0.00 & -2.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & -2.00 & 0.00 & 20.00 & -16.00 & -2.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & -2.00 & -16.00 & 20.00 & 0.00 & -2.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & -2.00 & 0.00 & 20.00 & -16.00 & -2.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & -2.00 & -16.00 & 20.00 & 2.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -2.00 & 2.00 & 16.00 \end{bmatrix}$$

Figure 3.c. Step-by-step illustration of matrix construction

Step 4. Construct the R.H.S. vector Rewrite equation (11) with the defined notation, we will have the right hand side vector equal to

$$\int_{I_j} f\,\phi_{k_v}^{(I_j)} + \underbrace{\{\phi'_{k_v}\}_j[\phi_{k_u}]_j}_{\text{symmetric term}} + \underbrace{\frac{\gamma}{|I_j|}[\phi_{k_u}]_j[\phi_{k_v}]_j}_{\text{penalty term}} \tag{29}$$

which is a 8×1 vector in our example. The integral term is computed with respect to intervals, which can be found explicitly or by Gaussian Quadrature with careful handle of the basis functions mapping from the master intervals to local intervals. For the symmetric term and the penalty term which are to be computed with respect to nodes, notice that we define the jump $[\phi_{k_u}]_j$ to be zero for all internal nodes since u is continuous. Hence we only need to consider those on boundary nodes x_0 and x_{N+1} and add them to the I_0 position and I_N position of the vector accordingly.

Notice that the jump $[\phi_{k_u}]_0$ is essentially $u(a)$ and $[\phi_{k_u}]_{N+1}$ is essentially $-u(b)$. With respect to **Note 1** in **Step 3**, we then obtain our boundary symmetric terms $\phi'_{k_v}(x_0^+)u(a)$ and $\phi'_{k_v}(x_{N+1}^-)\big(-u(b)\big)$, and also the boundary penalty terms $\frac{\gamma}{|I_0|}u(a)(\phi_{k_v}(x_0^+))$ and $\frac{\gamma}{|I_N|}\big(-u(b)\big)\big(-(\phi_{k_v}(x_{N+1}^-))\big)$. They are added to the I_0-th and I_N-th position of the vector. Substituting everything we would get our final R.H.S. vector

$$[0.25 \quad 0.25 \quad 0.25 \quad 0.25 \quad 0.25 \quad 0.25 \quad 4.25 \quad 16.25]^T$$

Step 5. Solve the linear system. With the given matrix and R.H.S. vector, we may now solve the linear system. In C programming, we may use the dense matrix solver *dgesv* in *LAPACK*, which implements the LU decomposition with partial pivoting and row interchanges:

```
dgesv_(size_Of_Global_Matrix, number_Of_RHS,
       pointer_To_Global_Matrix, leading_Dimension_Of_Global_Matrix,
       pivot_Indices,
       pointer_To_Solution, leading_Dimension_Of_Solution,
       info);
```

and our u_h approximation, illustrated by $Matlab$, would be as follow:

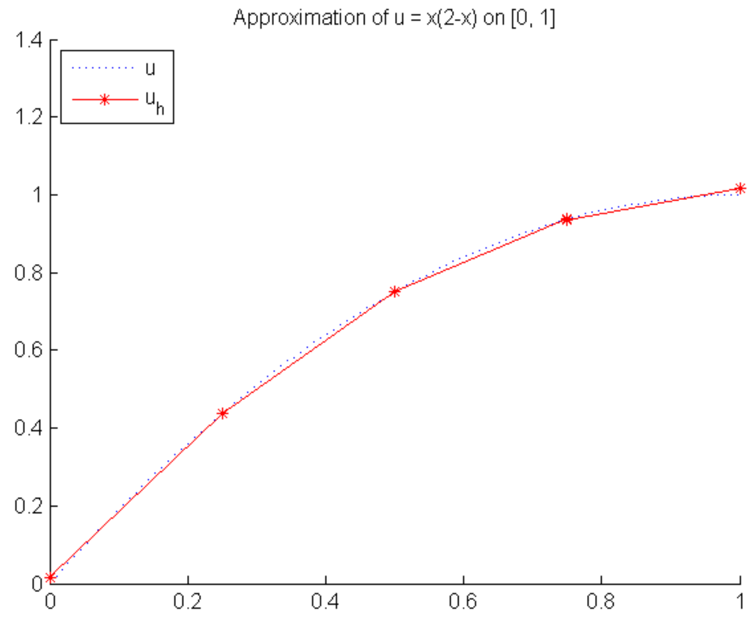


Figure 3.d. Approximation of u using DG-FEM.

4 Parallel Computation on 1D DG-FEM

Overview: *In this section, we are going to see how we may perform parallel computation on 1D DG-FEM: in the first stage we work on the linear system construction, while in the second stage we work on the linear system solver.*

4.1 Before we begin

It is essential to understand why we are interested in parallel computation on DG-FEM. In **Section 3** we have already seen the result of approximating u using 4 cells. We may further increase the number of cells to solve the same problem:

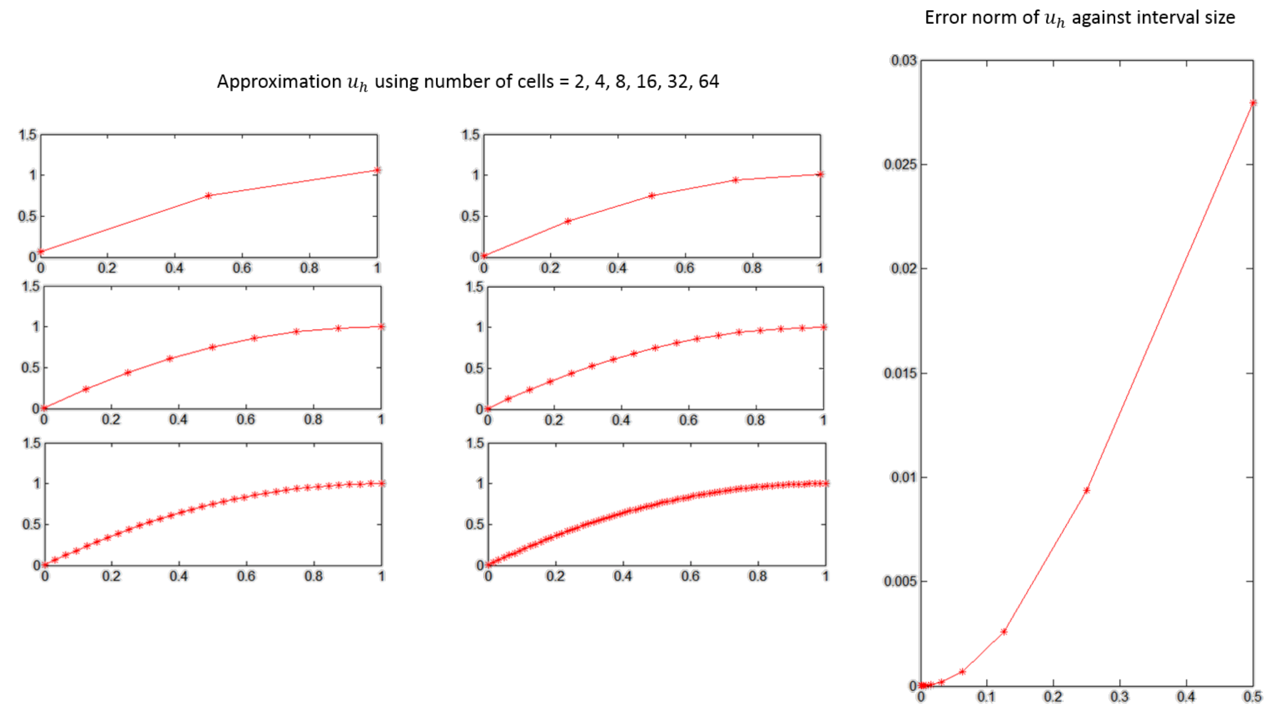


Figure 4.1: performance of u_h in increasing number of cells

We can see that the error norm $||u - u_h||_2 = \left(\sum_I \int_I (u - u_h)^2 \right)^{\frac{1}{2}}$ behaves as ch^2, where h denotes the cell size and decreases with the increase in number of cells. Hence the accuracy of our approximation is improving. However, larger degree of freedom follows and the matrix essentially grows larger. Time and memory becomes a huge concern in solving enormous linear system, therefore we wish to parallelize the computation: let multiple processors store information and work together **locally** to help reduce the overall computational time as well as the memory storage within a processor.

We would work in two stages: the first stage is to **construct** the linear system in parallel, while the second stage is to **solve** the linear system in parallel.

4.2 First Stage: Constructing Linear System in Parallel

In **Section 3**, **Step 3**, observe that each interval or node corresponds to a local matrix during the calculation, as shown in Figure 3.c.(for our convenience , we now call these local matrices at interval or node level to be **cell matrices**). Notice that the construction of R.H.S. vector works in a similar flow so we omit the writing here. Now it is natural to design parallelization in the following way:

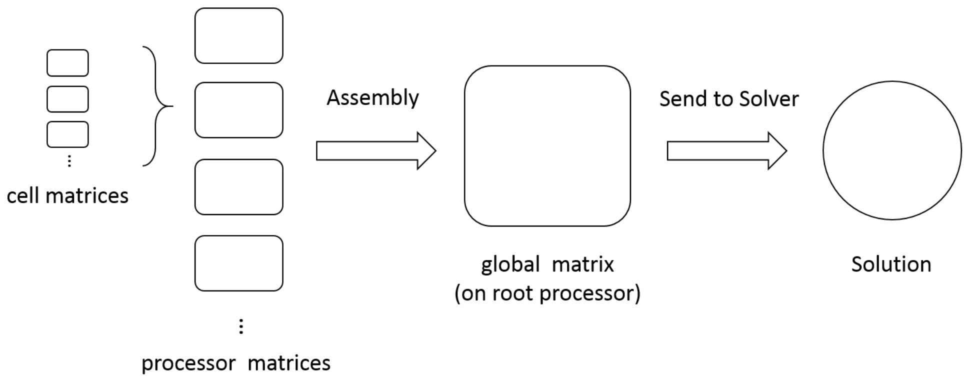


Figure 4.2.a: Matrix construction in parallel

We may work in the *MPI* environment to do the parallelization.

Step 1. Assign each processor to work on a certain number of **sequential** cells, do the **assembly** process and obtain a corresponding local matrix at the processor level (for our convenience, **processor matrix**). Since the calculation of cell matrices on the boundary nodes is different from those on the internal nodes (**Section 3**, **Note 1**), we may specify the root processor to handle the boundary cell matrices.

Step 2. Assign the root processor to receive processor matrices from other processors and do the **assembly** process to obtain the **global matrix**.
For the root:

```
MPI_Recv(pointer_To_Processor_Matrix, size_Of_Processor_Matrix,
         type, sender_ID, tag, MPI_COMM_WORLD, &status);
```

and for other processors:

```
MPI_Send(pointer_To_Processor_Matrix, size_Of_Processor_Matrix,
         type, root, sender_ID, tag, MPI_COMM_WORLD, &status);
```

Notice that the processor matrices may have overlapping positions in the global matrix, and the insertion and addition of the processor matrices to positions in the global matrix is to be determined and done by the root processor.

Step 3. After the construction of the global matrix, the root processor may proceed on sending the matrix to the solver as in **Section 3, Step 5**. Together with the R.H.S. vector we can then solve for the linear system.

Time Analysis. Enlarge the number of cells up to $10,000$, we may now see clearly the time reduction in linear system construction (including the *MPI* communication time):

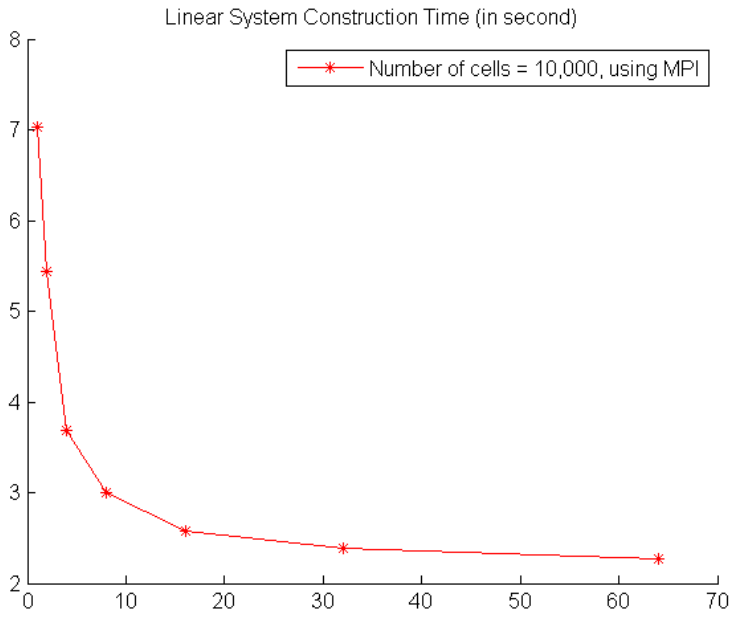


Figure 4.2.b: Linear system construction time against number of processors = 1, 2, 4, 8, 16, 32, 64

However, the overall computational time may still be large when considering also the time used by the linear system solver. For instance, in this example we use *dgesv* which takes up to around 20 seconds to complete the solution. In general, $LAPACK$ provides only direct methods – matrix factorizations as the solver option, yet the fact that our stiffness matrix is symmetric and positive-definite actually indicates that we may find some iterative methods, more specifically the **Conjugage Gradient Method**, very useful in reducing the computational cost in solving our large linear system.

Memory Improvement Notice that for each processor except the root, it only needs to handle the processor matrix and thus the memory usage for most of the processors individually would be small.

On the other hand, notice that in the above procedure we are storing every entry of the global matrix regardless of the fact that our matrix is sparse with non-zero entries confined to a diagonal band. In *LAPACK*, *dgbsv* takes care of input matrices in such structure(**band matrix**), and in general, matrix representation using **Compressed Row Storage** format is more desirable for sparse matrices. In such format, the assembly progress may require more careful handling as we need to specify a more complex local-to-global mapping, but the sequentiality of cells owned by a processor in **Step 1** of this section may also be relaxed since we may omit the zeros when we are storing the processor matrices with non-sequential cells.

4.3 Second Stage: Solving Linear System in Parallel

We now aim for a **fully parallel** code structure – that is, there is no communication between processors, so they work totally in parallel. Then we must get rid of the assembly process as well as consider parallelization in solving the linear system in addition to that in constructing the linear system.

Unlike direct methods which modifies the matrix as a whole to obtain the exact solution of the linear system in finitely many operations, iterative methods use successive approximations to obtain more accurate solutions to a linear system at each step, and enjoy the benefit that they involve the matrix only via the context of matrix-vector products (**MVP**) (c.f. [2]). Therefore, parallelization may be implemented in iterative methods by decomposition of the matrix and the solution vector into blocks (c.f. [3]).

In particular, the *Krylov subspace methods* in non-stationary iterative methods are usually chosen for parallelization due to dominance of MVP computations and independent vector updates (c.f. [4]). *AztecOO* in *Trilinos* provides implementation of parallelization using preconditioned Krylov methods, and thus comes to our choice as a solver package (c.f. [5]). The overall parallelization goes in the following way (the R.H.S. vector construction is omitted):

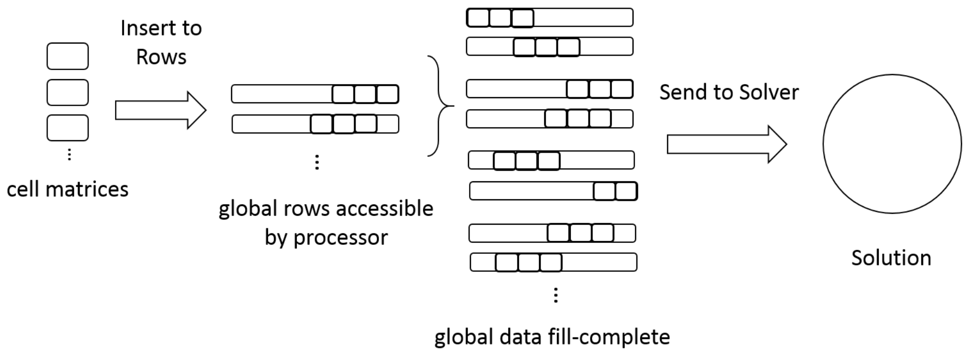

Insert to Rows
cell matrices
global rows accessible by processor
global data fill-complete
Send to Solver
Solution

Figure 4.3.a: Solving linear system in parallel

Step 1. Construct the **row map** – assign each processor a number of **global rows**, the data of which they may have access to set or modify:

```
Epetra_Map rowMap(sum_Of_Global_Rows,
                  number_Of_Local_Rows, global_Indices_Of_Local_Rows,
                  starting_Index, &comm);
```

Note that the rows need not to be sequential for each processor, while access to the same row for different processors should be avoided. The **column map**, which allows processors to have access to the column-entries in their assigned global rows, may be either manually constructed or automatically constructed after column entries are entered to the global rows.

Also note that we are no longer working on processor matrices as in **Section 4.2**, but instead working on accessible global rows by each processor. Here we say 'accessible' instead of 'owned' since although the processors access the global rows row-wisely, the actual memory distribution in *Epetra_CrsMatrix* that we are working on goes in an arbitrary two-dimensional distribution of rows and columns over processors.

Step 2. Assign each processor to work on a number of cells. The cell matrices of these cells should have entries located on the global rows that the processor has access to, so after computation of each cell matrix, the processor is able to insert or sum the values into the corresponding global rows position. The matrix is stored in **Compressed Row Storage** format.

```
Epetra_CrsMatrix *Global_Matrix
= new Epetra_CrsMatrix(Copy, rowMap, starting_Index);

Global_Matrix -> InsertGlobalValues
                (global_Rows_Index, number_Of_Entries,
                 values_Of_Entries, column_Indices_Of_Entries);

Global_Matrix -> SumIntoGlobalValues
                (global_Rows_Index, number_Of_Entries,
                 values_Of_Entries, column_Indices_Of_Entries);
```

Note that the cell matrices of adjacent cells are going to have values on overlapping global rows. However since each global row can only be accessible by one processor, in order for the summation of values on the overlapping global rows to be completed we must either introduce communication between processors, or use the **ghost cell** method: let each processor have additional cell information in order to complete their local work.

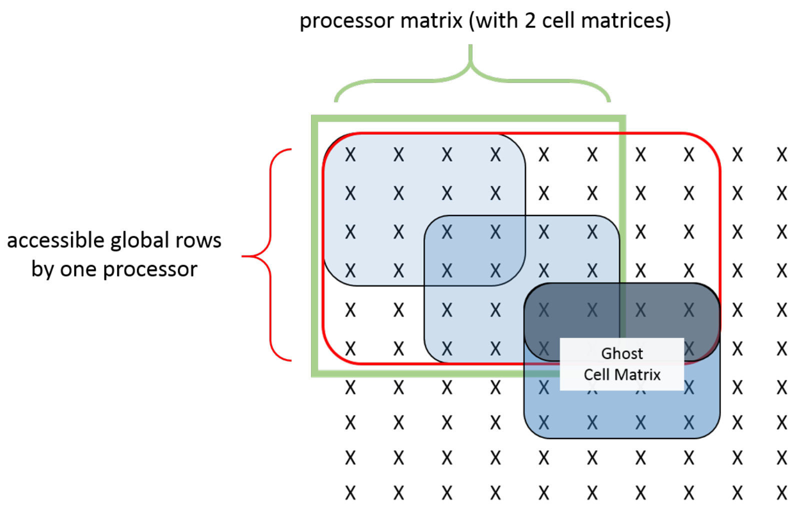

processor matrix (with 2 cell matrices)
accessible global rows by one processor
Ghost Cell Matrix

Figure 4.3.b: Ghost cell method

We may implement the ghost cell method by asking the processors to calculate also the ghost cell matrix, so as to have completed global rows. After all the entries are inserted, *fill-complete* is called to do local rearrangement of data via distributed-memory communication:

```
Global_Matrix -> FillComplete();
```

and the graph structure of the matrix is then fixed. Note that we may still modify the entry values afterwards, but set of the entries should be done before this command is called.

Step 3. The *AzetecOO* solver may now be created to solve the problem:

```
u = new Epetra_Vector(rowMap);
problem = new Epetra_LinearProblem(GlobalMatrix, u, RHS);

AztecOO solver(*problem);
AZ_defaults(options, params);
solver.SetAllAztecOptions(options);
solver.SetAllAztecOptions(params);
solver.Iterate(max_Number_Of_Iterations, tolerance);
```

Linear System Construction Time Analysis As discussed in **Step 2**, there is extra communication time between processors or extra ghost-cell-matrix calculation time for each processor in order to complete the insertion and summation of global entries. Nevertheless, when we are using the ghost cell method, the linear system construction time is hugely reduced compared to that in **Section 4.2**.

Number of Processors	1	2	4	8	16	32	64
using *MPI*	7.032s	5.440s	3.683s	2.995s	2.582s	2.385s	2.273s
using *Trilinos*	0.0148s	0.00796s	0.00434s	0.00255s	0.00194s	0.00191s	0.00295s

Figure 4.3.c: Comparison of linear system construction time using number of cells = 10,000

Indeed, we may compare the performance of *MPI* using number of cells equal to 10,000, with the performance of *Trilinos* using number of cells up to 5,000,000:

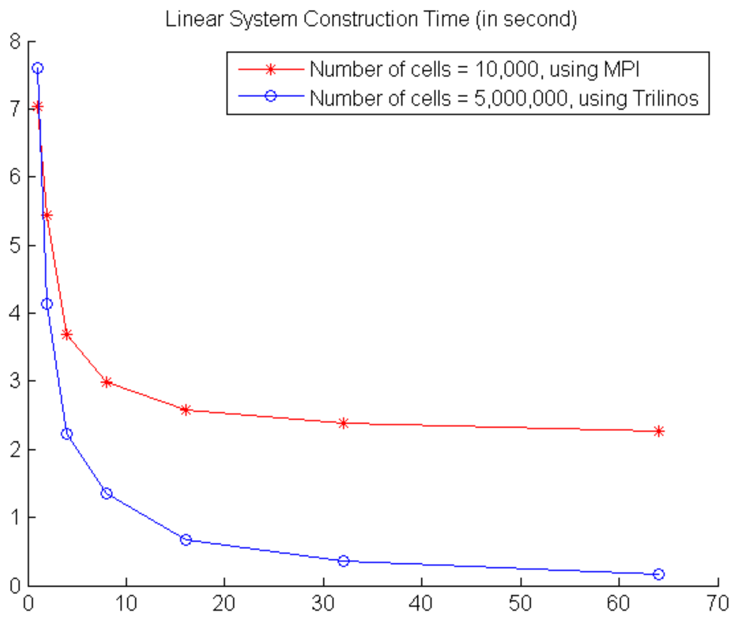


Figure 4.3.d: Comparison of linear system construction time against number of processors = 1, 2, 4, 8, 16, 32, 64

The main factor of the improvement may be the usage of Compressed Row Storage format that brings effective exchange of information from local to global. Also the entries are inserted to the global environment once they are computed at the cell matrix level, skipping the construction, sending and receiving of the processor matrices and thus result in less construction time.

Linear System Solver Time Analysis

Choice of Solvers. The parameters in *AzetecOO* may be specified. By default, the solver using is **Generalized Minimal Residue** (*GMRES*). However since our stiffness matrix enjoys symmetry and positive definiteness, we may set

```
options[AZ_solver] = AZ_cg;
```

to use **Conjugate Gradient** method in our problem.

Choice of Pre-conditioners. We may also consider the use of **pre-conditioners**, which works on transformation of the matrix to an equivalent form that has more favourable properties in an iterative method. Although using a pre-conditioner induces extra cost initially for the set up and per iteration for applying it, we may have a gain in convergence speed (c.f. [5]) The choice of pre-conditioners, just as the choice of solvers, depends on the properties of the original matrix. Here we choose incomplete factorizations and algebra multigrid pre-conditioners for performance comparison.

For pre-conditioner using incomplete factorizations (*ilu*), we may set:

```
options[AZ_precond] = AZ_dom_decomp;
options[AZ_subdomain_solve] = AZ_ilu;
```

For pre-conditioner using algebraic multigrid preconditioning, we may use the *ML* package on *Trilinos*:

```
Teuchos:: ParameterList MLList;

ML_Epetra::SetDefaults(ProblemType, MLList);
ML_Epetra::MultiLevelPreconditioner* MLPrec
         = new ML_Epetra::MultiLevelPreconditioner
           (*Global_Matrix, MLList, true);

solver.SetPrecOperator(MLPrec);
```

Algebraic multigrid preconditioning that typically works best on sparse positive definite matrices. The performance of *ML* may be tested by changing its parameters: *aggregation*, *coarse-type*, *smoother*, *increasing or decreasing*, *max level*, and more (c.f.[6]). For instance, if we set:

```
ML_Epetra::SetDefaults("DD", MLList);
MLList.set("aggregation: type", "Uncoupled");
MLList.set("coarse: type", "Amesos-UMFPACK");
MLList.set("smoother: type", "Aztec");
MLList.set("max levels", 6);
MLList.set("increasing or decreasing", "increasing");
```

Then *ML* may have a very desirable performance when compared to the *ilu* pre-conditioner:

	Number of Processors	1	2	4	8	16	32	64
Using *CG-ilu*	solving time	7.947s	5.222s	4.889s	4.867s	4.662s	4.130s	4.909s
Using *CG-ML*	solving time	11.818s	8.872s	6.223s	5.534s	2.912s	1.333s	0.723s
Using *CG-ilu*	number of iterations	2	9	24	40	68	128	249
Using *CG-ML*	number of iterations	2	6	9	10	11	10	12

Figure 4.3.e: Comparison of linear system solver using number of cells = 5,000,000

When we increase the number of processors, the solving time using *ML* is hugely decreased while the number of iterations has been relatively stable. Some more result regarding parameters of the *ML* pre-conditioner may be found in the **Appendix**. These figures provide insight to the question *how far we may reach* using the existing solver packages in solving the linear system.

5 Future Work

After implementation of parallelization on 1D DG-FEM, we may continue to work on the followings:

1. General DG-FEM. It is natural to think of expanding the 1D parallel code to the 2D and 3D case. From **Section 2.3** we can imagine that the code structure of the 2D case is going to be more complicated, as the code has to be able to handle information in three categories: **cells**(*triangles*), **edges**(*internal* or *boundary*), and **nodes**, whereas in the 1D case we need only to handle two: cells (*intervals*) and nodes (*internal* or *boundary*). Also, while we have the jump conditions being stored on the sub-diagonal blocks of the matrix in the 1D case, we cannot do so in the 2D case given the fact that now each cell comes across three other cells instead of two – so the linear system construction is going to be more complex. The same applies to the 3D case, and the design of parallelization on the 2D or 3D cases would be flavourable.

2. Adaptive meshing. The discontinuity of the cell boundary solutions provides effective means for local refinement. As the computation is naturally cell-oriented, a change in conditions within the cell, such as higher order or discretisation of the geometry so as to achieve improve local accuracy, may be easily achieved in DG-FEM. Therefore we may look into expanding the code to be adaptive.

3. More applications. The Poisson's equation is an example that is relatively easy to work on, since it involves only a derivative term. When come to chemical transport phenomena, most often more derivative terms and physical quantities such as time, mass, momentum or energy are involved. Therefore expanding the code to cover different chemical transport equations may be considered.

Appendix

The result of solving the example in **Section 4.3** using CG solver and *ML* pre-conditioner with the following parameters:

- Number of cells : 5,000,000
- Degree of freedom: 2
- ML pre-conditioner option: 'DD' (domain-decomposition method)
- ML pre-conditioner option: 'increasing'
- ML pre-conditioner option: 'max levels: 6'

Aggregation	Coarse type	Smoother	No. of Procs	1	2	4	8	16	32	64
Uncoupled	Amesos-KLU	Aztec	Time(s)	11.78661	8.89994	6.21667	5.49554	2.90557	1.33267	1.12753
			No. of Iter.	2	6	9	10	11	10	21
MIS			Time(s)	11.77905	8.81326	7.02514	7.13098	4.07430	18.15375	1.12629
			No. of Iter.	2	6	11	14	17	187	21
METIS			Time(s)	9.43952	7.99520	11.42985	14.72198	13.26475	12.55926	11.19096
			No. of Iter.	2	8	29	43	81	157	296
ParMETIS			Time(s)	9.03556	8.53104	9.50912	13.79076	12.16044	11.70670	11.97191
			No. of Iter.	2	10	23	40	74	146	318
Uncoupled		Gauss-Seidel	Time(s)	KILLED	KILLED	KILLED	-	-	-	-
			No. of Iter.	>531	>935	>1574	-	-	-	-
		Jacobi	Time(s)	148.23950	82.25089	48.72048	44.53721	22.08759	10.51612	4.98469
			No. of Iter.	149	150	150	150	150	145	146
MIS			Time(s)	142.65031	78.75227	45.62901	43.18269	21.41529	10.97140	4.98940
			No. of Iter.	149	144	146	145	145	150	145
METIS			Time(s)	KILLED	KILLED	KILLED	KILLED	KILLED	KILLED	KILLED
			No. of Iter.	>801	>1535	>2618	>2700	>5191	>10566	>22088
ParMETIS			Time(s)	KILLED	KILLED	KILLED	KILLED	KILLED	KILLED	KILLED
			No. of Iter.	>932	>1615	>2598	>2736	>5477	>10351	>22315
Uncoupled	UMFPACK	Aztec	Time(s)	11.81769	8.87170	6.22301	5.53356	2.91172	1.33327	0.72338
			No. of Iter.	2	6	9	10	11	10	12
MIS			Time(s)	11.77558	8.76869	7.04594	7.11715	4.07191	18.16983	1.12752
			No. of Iter.	2	6	11	14	17	187	21
ParMETIS			Time(s)	9.03989	8.52426	9.50146	13.72986	12.18949	11.72245	11.99328
			No. of Iter.	2	10	23	40	74	146	318
Uncoupled		Jacobi	Time(s)	147.68294	82.32959	48.67825	44.51618	22.06348	10.55785	4.98232
			No. of Iter.	149	150	150	150	150	145	146
	Superludist	Aztec	Time(s)	11.79255	8.93527	6.24897	5.56728	2.94533	1.37276	0.77184
			No. of Iter.	2	6	9	10	11	10	12
	MUMPS		Time(s)	11.78701	8.91217	6.21046	5.58920	2.95156	1.39642	0.81123
			No. of Iter.	2	6	9	10	11	10	12
	Jacobi		Time(s)	11.69444	11.73216	11.15663	17.72796	16.07835	10.59748	12.48447
			No. of Iter.	2	10	21	41	79	108	272
	Gauss-Seidel		Time(s)	11.80211	11.00963	11.17005	16.90885	16.06955	11.95901	11.31063
			No. of Iter.	2	9	21	39	79	121	246

Reference

[1] Sven Berger: Introduction to discontinuous Galerkin element methods, 1, Institute of Aerodynamics, 2003.

[2] Rashid Mehmood, Jon Crowcroft: Parallel iterative solution method for large sparse linear equation systems, 2.1, University of Cambridge, UCAM-CL-TR-650 ISSN 1476-2986, 2005.

[3] Rashid Mehmood, Jon Crowcroft: Parallel iterative solution method for large sparse linear equation systems, 4, University of Cambridge, UCAM-CL-TR-650 ISSN 1476-2986, 2005.

[4] Rashid Mehmood, Jon Crowcroft: Parallel iterative solution method for large sparse linear equation systems, 2.4, University of Cambridge, UCAM-CL-TR-650 ISSN 1476-2986, 2005.

[5] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 3.1.1, the Society of Industrial and Applied Mathematics, 2nd edition.

[6] Marzio Sala, Michael A. Heroux, David M. Day, James M. Willenbring: Trilinos Tutorial, 11.4, Sandia Report SAND2004-2189, 2010.

Acknowledgement

This project was sponsored by Oak Ridge National Laboratory, Joint Institute for Computational Sciences, University of Tennessee, Knoxville and The Chinese University of Hong Kong. It is my great honor to be financially supported by the aforementioned institutions to work on this project in the past ten weeks.

A sincere thank to my mentors, Dr. Ohannes Karakashian and Dr. Kwai Wong, for their continuous support and guidance throughout the project. Dr. Karakashian has been more than generous in offering intellectual lectures that are interesting and informative, while Dr. Wong has been very considerate and full of wisdom in providing prospective insights to problem solving both at work and in daily life.

Also thank the CSURE participants for being so cheerful and helpful in work and in play. Especially thank Ming Wong and Helsa Chan for being such two lovely flatemates who often bring laughter and good food to the apartment. The memory in Apt. 102, Block 10 is unforgetable.

Lastly, thank my parents and my boyfriend for their love and endless support throughout the time. Thank God for His grace in letting me have such an invaluable experience.