SiSc Lab
Unstructured Finite Element Method

Klim Kolyvanov[†], Philip Kenny
[†]klim.kolyvanov@gmail.com

February 6, 2015

Abstract

In this project we are solving 2D-(Un)steady heat diffusion equation using finite element method. Both sequential and parallel implementations will be discussed.

1 Introduction

Finite element method (FEM) is a numerical technique for finding approximate solutions to boundary value problem for partial differential equations.

It uses subdivision of a whole problem domain into simpler parts, called finite elements, and variational methods from the calculus of variations to solve the problem by minimizing an associated error function.

FEM collects many simple element equations over many small subdomains, named finite elements, to approximate a more complex equation over a larger domain.

The subdivision of a whole domain into simpler parts has several advantages:[3]

1. Accurate representation of complex geometry;
2. Inclusion of non-similar material properties;
3. Easy representation of the total solution;
4. Capture of local effects.

2 Mathematical statement

The heat distribution over time in a given region can be described by heat equation (1).

$$\begin{aligned} c_p\rho\frac{\partial T}{\partial t} - k\frac{\partial^2 T}{\partial x^2} &= f \qquad \text{for} \quad x \in \Omega, \\ T = \alpha\frac{\partial T_N}{\partial x} &+ \beta T_D \qquad \text{for} \quad x \in \partial\Omega, \\ T &= T_0 \qquad \text{for} \quad t = 0. \end{aligned} \tag{1}$$

where c_p is a heat capacity under constant pressure, ρ is the density and f is a source term.

Depending on values α and β we can get Dirichlet ($\alpha = 0$), Neumann ($\beta = 0$) or Robin ($\alpha \neq 0$, $\beta \neq 0$) boundary conditions.

The equation (1) is, in general, not solvable analyticaly for any source term f. That is the reason, we have to switch to another method.

In our case that is finite element method.

The main idea of the finite element is to divide the domain Ω into small parts (elements) and solve the arising equation for each element. However, that is not that trivial.

We start with the initial equation (1) and multiply it by the weighting (test) finction. After that we have to integrate the equation over the whole domain Ω.

That is called the weak formulation (see equation (2)) of eqution (1).

$$\begin{aligned} \int_\Omega \left(c_p\rho w\frac{\partial T}{\partial t} + k\frac{\mathrm{d}w}{\mathrm{d}x}\frac{\partial T}{\partial x} \right) \mathrm{d}x = \int_\Omega wf \, \mathrm{d}x + \\ \int_{\partial\Omega} w \left(k\frac{\partial T}{\partial x} n_x \right) \mathrm{d}x \end{aligned} \tag{2}$$

Depending on the way we choose weighting functions w, we get different FEM formulations. We first suppose that the solution (temperature) for the problem (1) has the form

$$T(x,y) = \sum_{j=1}^{nn} T_j S_j(x,y) \tag{3}$$

where $S_j(x,y)$ the so-called shape functions.

To use Galerkin formulation, we have to take $w = S_j(x,y)$.

As a next step, we plug in both $T(x,y) = \sum_{j=1}^{nn} T_j S_j(x,y)$ and $w = S_j(x,y)$ in equation (2).

After some simplifications, we end up with

$$c_p\rho\frac{\mathrm{d}T_j}{\mathrm{d}t}\sum_{j=1}^{nn}\int_\Omega S_iS_j\mathrm{d}x + kT_j\sum_{j=1}^{nn}\int_\Omega \frac{\mathrm{d}S_j}{\mathrm{d}x}\frac{\mathrm{d}S_i}{\mathrm{d}x}\mathrm{d}x = \int_\Omega S_i f\mathrm{d}x + \int_{\partial\Omega} S_i k\frac{\partial T}{\partial x}n_x\mathrm{d}x \tag{4}$$

The equation (4) is nothing else but system of differential equations

$$[M]\dot{T} + [K]T = F + B \tag{5}$$

which can be further reduced by approximating the time derivative using forward Euler method (explicit scheme)

$$\left.\frac{\mathrm{d}T}{\mathrm{d}t}\right|_s = \frac{T^{s+1} - T^s}{\Delta t} + O(\Delta t) \tag{6}$$

$$[M]T^{s+1} = M]T^s + \Delta t(F + B - [K]T) \tag{7}$$

Since solving the full (in a sense of sparcity) system is expensive we would like to introduce the method that reduces the amount of time we spend on soving system of linear equations.

The idea is to "lump" the mass matrix. The mass matrix becomes diagonal and such a linear system can be easily solved

$$M_{L_{ii}} = M_{C_{ii}}\frac{\sum_{i=1}^n\sum_{j=1}^n M_{c_{ij}}}{\sum_{j=1}^n M_{c_{jj}}} \tag{8}$$

We will use the approaches introduced in this section further.

3 Serial implementation

The implementation of the finite element solution is programmed in C++. It is divided in to 4 main classes named Gauss, Elements, Nodes and Solver. The Gauss class defines the gauss points and weights required to carry out gauss quadrature on the elements as well as the shape functions used to interpolate the solution values. The Gauss class in never instansiated directly but it is the parent class of Elements, all its member functions and data members are inherited by Elements when it is declared.

Solver is a class that contains all the solvers for semi-discrete time discretisation. In both the serial and parallel code, only one solver is implemented. The solver, named explicit solver uses forward Euler to calculate the temperature values of the next time step. This function takes as an input the the Nodes and Elements object which represent the mesh and implements a while loop that continues until reaching a simulation time, iteration number or convergence limit. This loop solves one iteration of the linear system in cycle, equivalent to one time step then it updates the nodal temperature value.

The heart of the program is the Nodes and Elements classes. These define the data structures used to store the mesh as well as member functions to manipulate the values. Nodes holds `double` arrays with length equal to the number of nodes containing the x and y coordinates of the nodes, ordered in their global numbering system as well as the `double` array containing the initial temperature values that will be updated in the time loop. Elements contains the data members it inherits from Gauss as well as an `int` array of length number of element nodes multiplied by number of elements. This array stores the connectivity, which is an integer 'handle' to the global node number. This allows loops through elements to access the nodal data stored in the Nodes object.

In the serial code, most of the algorithm is contained in the Elements member functions. Most significant are the `void calculateJacobian(node* nodes)`, `void element::calculateElementMatrices(setting* settings, node* nodes)` `void element::applyBoundaryConditions(setting* settings, node* nodes)` and `void element::calculateRHS(node* nodes, setting* settings)`.

The function `void calculateJacobian(node* nodes)` reads coordinate values from the Nodes object and calculates the integral over each element from this the Jacobian for the transformation from reference element to physical element is evaluated. The determinant is stored and used to calculate the derivatives of shape functions with respect to the x and y coordinates of physical space.

`void element::calculateElementMatrices(setting* settings, node* nodes)` uses the previously calculated and stored derivatives and determinant to calculate the element-wise stiffness and mass matrices. It also sets the forcing terms for every node.

Function `void element::applyBoundaryConditions(setting* settings, node* nodes)` consists of two main loops. Firstly, the implementation loops through every node of each element and determines the appropriate face group to which to assign it. At this stage checking is carried out to ensure that only ‘safe’ overwrites of face groups occur, that is that every node is assigned to exactly one face group and that no impossible conditions are enforced, for example a Robin condition may cause a nodal temperature to be a function of itself and it’s neigbouring node, if the neighbouring node is assigned a Dirichlet condition then the first node’s temperature will become a monotonically increasing (or decreasing) function and the global solution will never converge.

The fourth important member function of Elements is `void element::calculateRHS(node* nodes, setting* settings)`. It calculates the ‘Righthand Side’ of the linear system to be solved at each time step. This must be computed every time step because it is a function of the present solution (temperature) values which are being recomputed in the time loop.

As mentioned the largest portion of coding was in the Elements object class. In particular ensuring that the boundary conditions do not corrupt the physical problem was complicated by the many edge cases, nodes that are shared by internal and external faces and nodes at the interface between face groups.

4 Solver validation

We used two cases to validate that our solver was getting the right values as the result.

The first one is a cylinder with a hole inside.[1] Due to symmetry we were using two co-centric meshed circles (see fig. (1)).

In this case, we were checking the implementation of our Dirichlet boundary conditions and the way the Jacobian computation works.

We prescribe Dirichlet boundary conditions on both inner and outer circles, basically, we keep the temperature constant on both boundaries.

First of all, we have to notice that the problem is symmetric. Therefore, the solution should look the same in all directions (For the dynamics see the circle.mp4 file attached).

In the fig. (2) the results for FEM and exact (analytical) solutions are given. The result, we have found, is quite accurate and the error is below 1 % ($\|e\|_{L_2} \leq 0.005$).

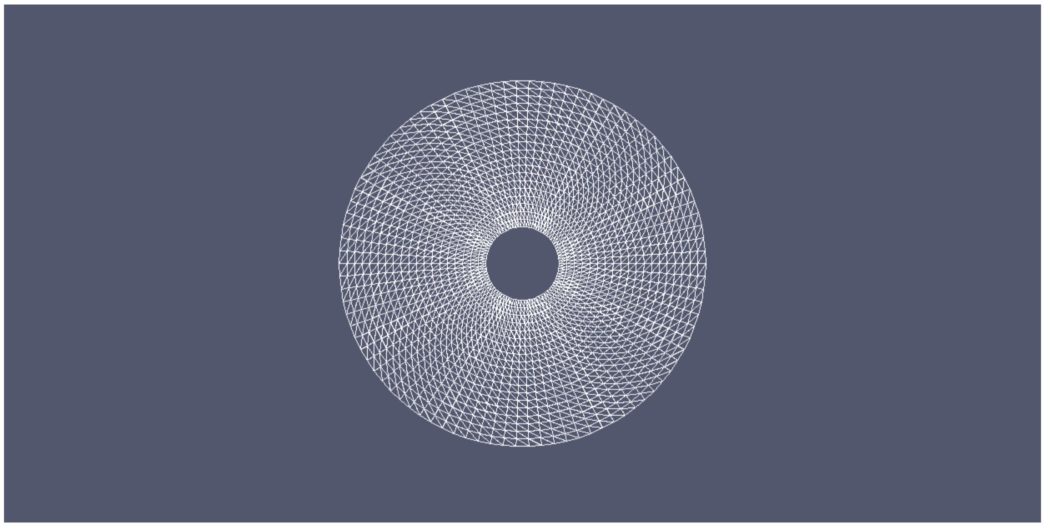
Figure 1. Cylinder mesh

The second validation case was Rod.[2] Here we wanted to check out Neumann boundary conditions implementation.

At the top, bottom and right boundary we set 0 Neumann conditions (That corresponds to insulated boundary) and on the left boundary we set Neumann condition equal to 1, prescribing the incoming heat flux.

The mesh for this case is given in fig. (3). The results (both analytical solution and our FEM solution) are given in fig. (4).

One can notice that the results are quite inaccurate ($\|e\|_{L_2} \leq 0.16$). That error can be caused by the mesh we used for the computations (it was quite coarse) and also by the way we apply BC (every time we have to choose which BC type to keep for current node if it shares both of them).

5 Parallel implementation

The parallel implementation of the finite element solver contains exactly the same classes and data structures as the serial implementation. The differences are in the functions that manage communication and domain decomposition. The member functions in Nodes and Elements that read in the

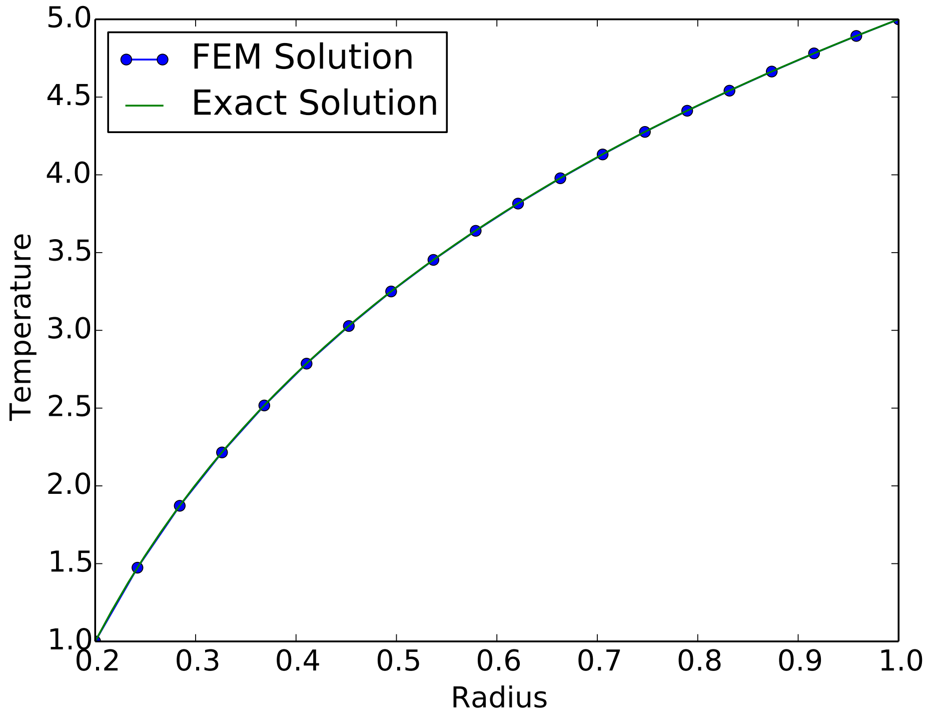

Figure 2. Cylinder solution

Figure 3. Rod mesh

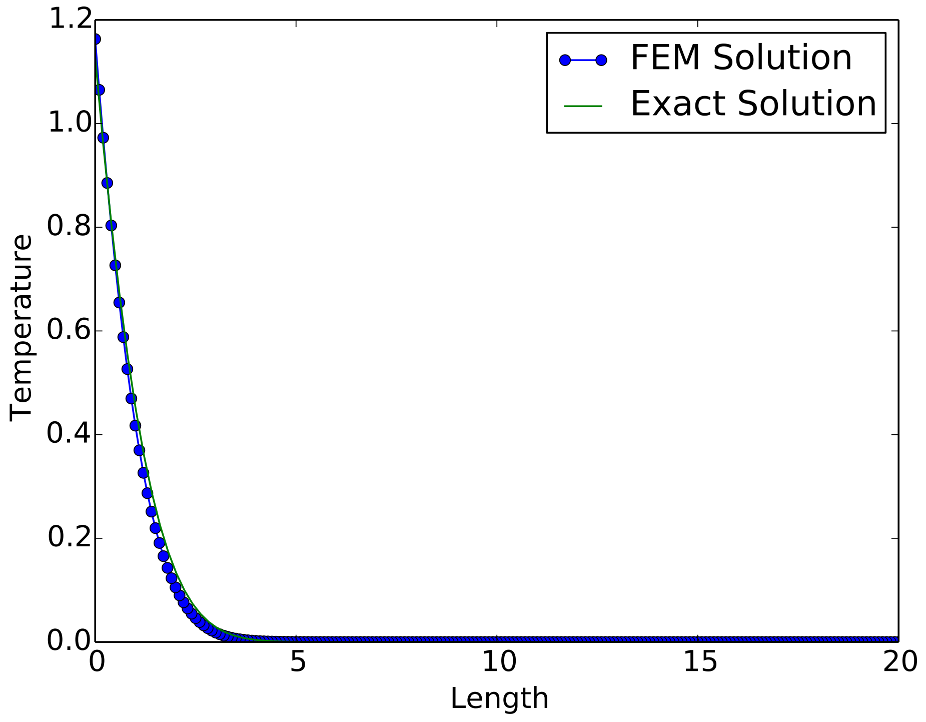

Figure 4. Rod solution

mesh data files are modified to read in an equal chunk of the `mxyz` `mien` and `mrng` files.

Additionally the Elements class contains functions `void element::calculateOffsets(node * nodes)` and `void element::modifyConn(node * nodes)`. `void element::calculateOffsets(node * nodes)` loops through all element nodes and determines which remote process they are stored on, then this information is stored in an integer array called `offsets` in the Nodes object which is used to determine which process and which memory address to use in communication.

Communication was implemented using MPI Message Passing Interface. In particular One-sided MPI aka Remote Memory Access was used. MPI is a library and API that enables process with separate memory address space to send and receive data from other processes by passing 'messages' which contain the data. One sided MPI works by allowing a group of processes to assign a portion of their local memory to a data 'Window' this window into memory is available to be accessed by remote processes. The exact way that this is implemented inside the MPI specification is quite complicated, but from a programmers point of view, it allows the use of functions that resemble the public 'getters' and 'setters' used in object oriented programming to access private data members. In this case however, data is not simply protected by being declared as 'private' but it is stored on a separate machine.

The specific function calls available are `MPI_Get()` `MPI_Put()` and `MPI_Accumulate`. `MPI_Get()` acts in such a way as to 'take' data from a remote process and store in on the local machine. `MPI_Put()` is the compliment, it takes local memory and 'writes' it to memory on a remote machine. `MPI_Accumulate()` is very similar to `MPI_Put()` except that it is associated with a reduction operator (SUM, MIN, MAX, XOR, etc) to allow more sophisticated communication rather than simply overwriting the data stored in the remote window.

In our implementation `MPI_Get()` is used to distribute nodal values to the processes that require it as part of their element-wise calculations. `MPI_Accumulate()` is used to return calculated values from the process that calculates them to the process that 'owns' the node. Different reduction operators are used in different situations, for example temperature contributions from different elements need to be summed back to the process that owns them, whereas face groups must overwrite each other as every node must belong to exactly one face group.

In order to speed up communication, MPI data types are also used. MPI data types are custom data types that can have arbitrary size and structure, data does not have to be contiguous and it can contain many different primitive data types such as `char, int, double`. The advantage in using MPI data types is that, since we have previously calculated the offsets of the various values we require from remote processes, we can create a custom data type that includes only these values and ignores data stored between these offsets. This allows a single MPI call to be made with a single data type transferred which includes all requisite data. A second advantage is that by using different data types for the remote and local representations, it is possible to reorder data 'enroute' between processes. This was used to take sparse data (with plenty of offsets) from remote processes and store it in a contiguous block of memory on the local machine. It similarly makes the reverse communication back to the remote process much simpler as well.

Challenges with the parallel code are very similar to the serial version, the complication of remote and local representation and degenerate values (each process that shares a node has a local version stored in local machine memory) simply adds another layer of difficulty to the already difficult parts. Specifically boundary conditions, now the problem of overwriting boundary conditions is not as simple, a process cannot simply invoke a conditional statement to decide if overwriting the current face group is acceptable because the information about other applicable face groups can be stored on a remote process. Nor is the communication of these values simple, for other variables (temperature, mass) a sum or replace operator can be used depending on the requirements. Face group requirements are much more complicated, a node that already has a face group set must ensure it is only overwritten if the face group overwriting it is a reasonable choice, (for example don't overwrite a face group associated with a boundary condition with `0`, signifying an interior face. Unfortunately the face group itself is just an integer handle to an array of face groups that then specifies what type of boundary condition it represents. Even if the type of boundary condition can be determined, some 'overwrite pairs' are acceptable (Dirichlet and Neumann by anything else but Robin only by Robin). `MPI_Accumulate()` does not allow custom reduction operators to be used and only a selection of standard reduction operators are available.

In our implementation we settled for the `MPI_MAX` reduction, this prevents interior faces (type `0`) from overwriting previously assigned boundaries but does nothing to prevent, for example Dirichlet face groups from overwriting

Robin. This limits the robustness of the code and requires careful labeling and assignment of face groups to faces to prevent erroneous overwrites.

6 Performance

The performance obtained for the parallel implementation varied widely. The first attempt at parallelisation using the original data numbering produced very poor scaling (see fig. (5)). Very low speed up was achieved up to 32 processes (reaching a maximum of 4.XX) beyond this execution time actually took longer. In this case the cause of poor performance is the data locality.

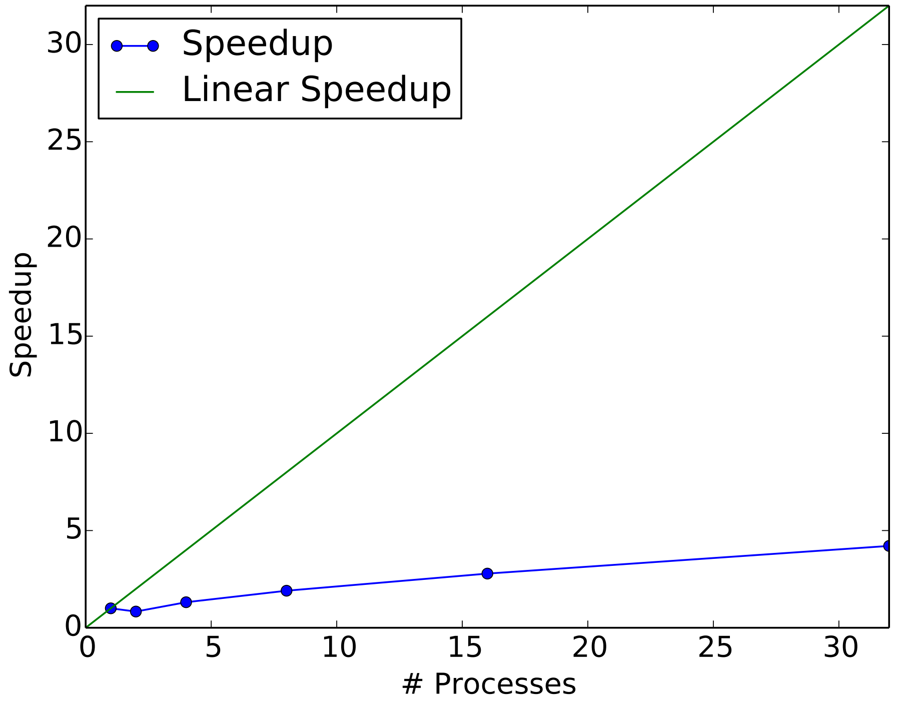

Figure 5. “Strong” scaling

Fundamentally, the problem starts with the ‘dual’ representation of the mesh. The complete mesh is stored in two separate data structures. The

Node class stores values associated with the nodes, this is the geometry. Values include the coordinates in physical space, and the solution (temperature) values. The Element class stores the connectivity of these nodes, the topology. To perform analysis on the mesh it is necessary to have a complete representation including all nodes and all the connectivity belonging to a the domain or subdomain of interest.

The naive approach used in out initial parallelisation took no account of the relationship between nodes and elements. The original, unmodified input files use an arbitrary numbering of nodes and elements, this ignorance of the physical domain is propagated into the parallel read and write functions in our `Elements` and `Nodes` classes. Because of this many nodes required for computation of the local elements are stored on remote processes and many of the locally stored nodes are unneeded for local computation. This situation is exacerbated by increasing the number of processes, because each process has a smaller number of elements and nodes in the same sized domain, the chance of overlapping nodes and elements is reduced.

In fact there are two problems at work here. An arbitrary reading of elements gives no guarantee that the elements will comer some nicely compact subdomain (or even that the elements are physically contiguous) this means that not only is more communication required (for increased interface between subdomains) but also that more memory is required to store nodes because fewer nodes are shared by local elements. So this increases both communication overhead, memory use and memory read/writes.

The second problem is that the arbitrary numbering of nodes reduces the chance that local nodes are part of the locally stored elements. Again this increases memory use because all processes have the same number of locally stored nodes, data for every additional remote node that is required must also be stored locally. This requires additional communication. Again communication, memory use and read/writes are increased.

Both these complications occur to some extent at the same time and have a multiplicative impact on performance.

The solution to this problem is to use a renumbering of nodes and elements that accounts for the physical relationship between both parts of the mesh. This was achieved using permutation files created by a program called partition. THese permutation files provide a new order for nodes and elements to give a better domain decomposition.

Parallel implementation of this reordering is complicated by our use of One-sided MPI. One-sided MPI is useful when the process making the MPI calls knows the location of the remote data it needs to access, but the remote process has no knowledge of processes that need to receive data. During reordering the algebraic definition of which process contains which nodes is violated. This means that an initial round of communication is required to disseminate information on the ‘ownership’ of nodal data. This is precisely the step avoided in our implementation by using one-sided MPI and large portions of code would have to be rewritten to incorporate this new approach.

In order to make use of the permutation without rewriting the code, we created a new, serial program to permute the input files. Then the parallel FEM implementation remains unchanged and we simply read the new, permuted files in lieu of the original files. The results achieved with these new, permuted files is shown in fig. (6). Interestingly the parallel implementation now exhibits superlinear speedup up to a certain number of processes. We have no provable explanation of this but it seems likely that the use of more processes alleviates a memory bandwidth bottleneck as each process now has fewer memory read/writes for a given problem. Another interesting feature is the fact that for large numbers of processes improves for increasing simulation time (number of iterations of the time loop). This suggests that the primary cause of poor performance with many processes is the construction of the matrices and the boundary conditions rather than the calculations in the time loop.

One final feature obvious from the fig. (6) is that beyond 64 processes, we observe a decrease in speedup i.e. increasing the number of processes actually increases the execution time. This can be explained by a combination of the usual saturation experienced by parallel programs described by Amdahl’s law, where execution time cannot get shorter than the execution time of serial portions of the code and a violation of the permutation created by the partition program.

This violation is a result of our unchanged parallel code reading equal portions of the permuted input files. The permutation produced by partition is actually optimised for non equal partitioning of nodes and elements. The number of nodes and elements for optimal partitioning are provided at the end of the permutation files. These are blatantly ignored by our implementation. For smaller numbers of processes the effect of this suboptimal distribution is less noticeable. For larger numbers of processes, the disparity is more noticeable because the same variation from equal partitioning represents a larger fraction of the total local nodes/elements.

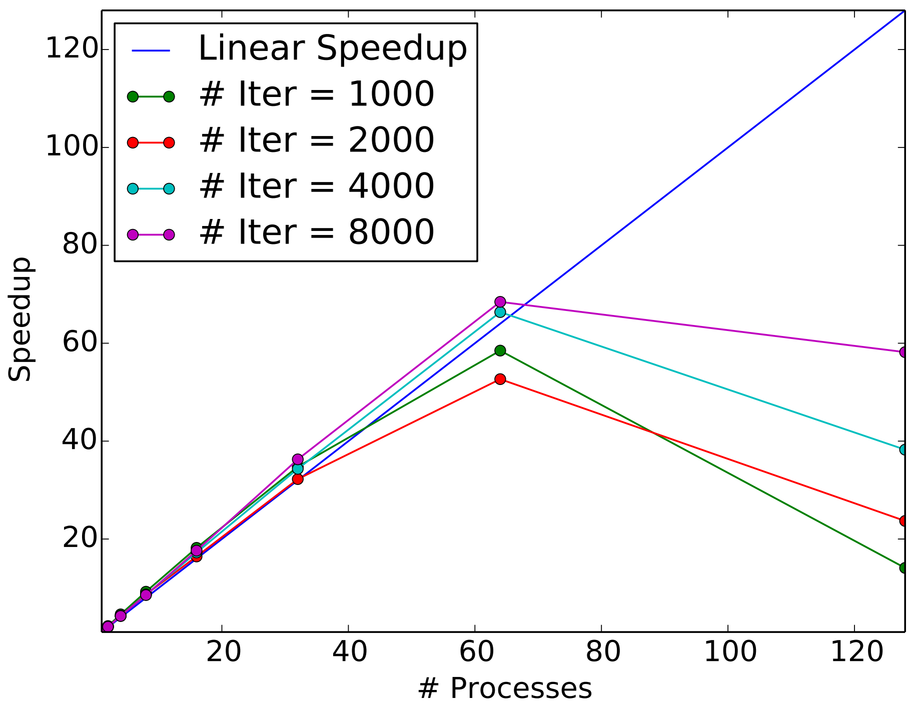

Figure 6. Strong scaling

7 Final Project (Real world problem): microchannel

A heat sink for cooling computer chips, which is fabricated from copper with machined microchannels, is shown in fig. (7). Within these microchannels, water flows and carries away the heat dissipated by the chips. The lower side of the sink is insulated while the upper side is exposed to ambient air When we consider the symmetric unit on the right side of the fig. (7), it is possible to treat the right and left surfaces as insulated boundaries.

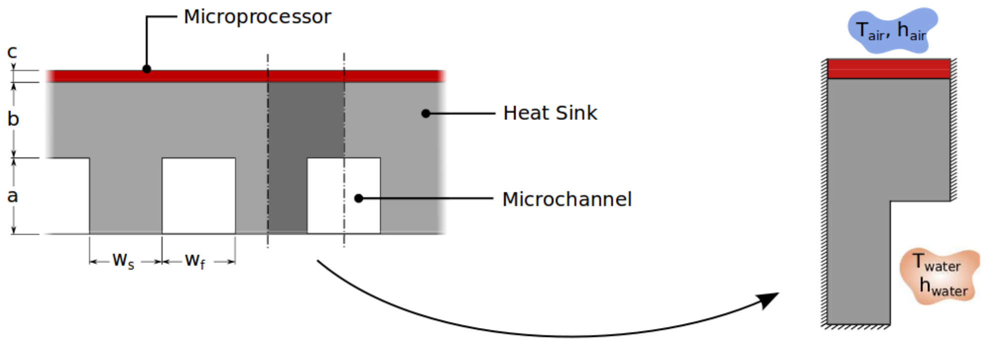

Figure 7. Illustration of the heat sink and the symmetric region

The two main questions were:

- Determine the maximum possible heat generation rate ($T < 75°\text{C}$);
- Determine the steady state distribution and time required to reach it.

The mesh for the microchannel is given in fig. (8).
The steady state solution was reached in $T_{sD} = 2.258 \cdot 10^{-4}$ seconds.
While the maximum heat generation rate was $\approx 2857[J/(mm^2 \cdot sec)]$.
The fig. (9) shows the microchannel at the steady state.

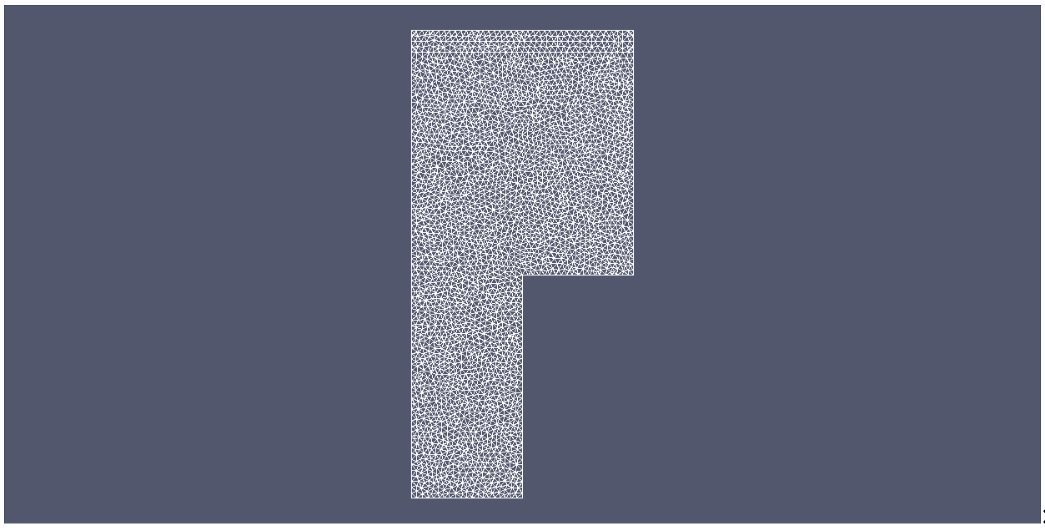
;

Figure 8. Mesh microchannel

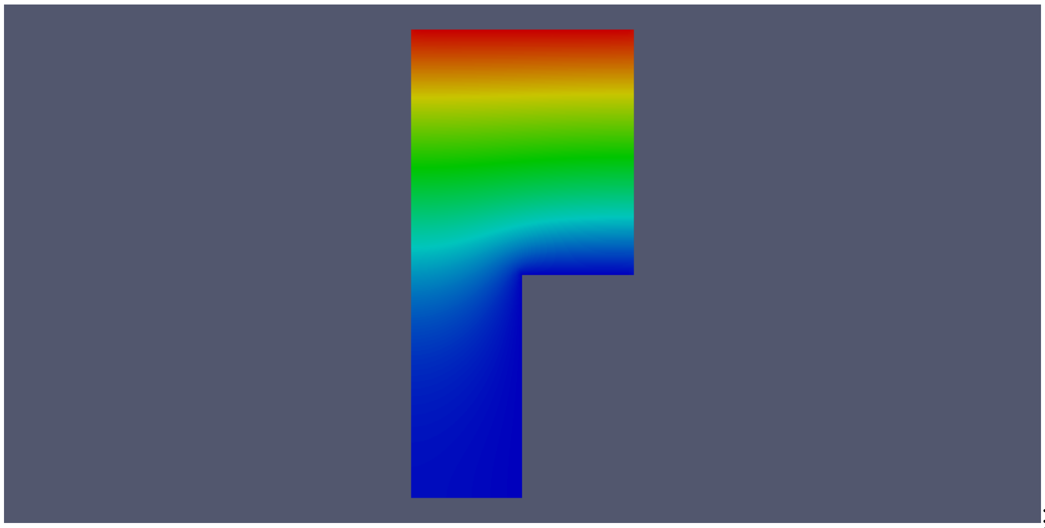
;

Figure 9. Microchannel solution

8 Future development

There are a number of improvements that can be implemented to either enhance the parallelism of the code or to extend its functionality. Firstly the optimal permutation should be implemented as discussed earlier this will require reworking a significant portion of the parallel code. This is a significant amount of work, but the parallelism and therefore the utility of this code is limited by this step, so this must be implemented and would have been if there had been more time for the project.

Secondly the use of some multithreading to take advantage of modern multicore architecture. OpenMP is the simplest and easiest to add to existing code so is a good starting point. We did conduct some experimentation with OpenMP but found it did not improve execution time, this may be an artefact of the memory bandwidth limitation, but more work is required to prove this. Even if loop parallelism is not possible it should be possible to use tasking to hide some of the communication overhead, one thread can start calculating results for the local nodes (which with correct permutation should be the majority of nodes) while a second thread invokes the MPI communication calls.

A third possibility for improving parallel performance is using a less strict memory managment paradigm. In the current program, MPI windows are used to facilitate remote memory access. `MPI_Win_fence()` is called to ensure memory consistency between acesses. This is a very strict interpretation and assuming that memory accesses are read only or writing to different memory addresses it is possible to relax these calls to `MPI_Win_fence()`. `MPI_Win_fence()` includes an implicit barrier and memory fence, using a more general form could allow for less synchronisation or at least reduce the impact of synchronising processes at every MPI call.

In addition to performance improvements, enhanced functionality would also increase the utility of the program. For instance, the explict forward Euler time discretisation is easy to implement but can prove to be very unstable. It is also, in general, not possible to choose a stable timestep without some experimentation. Also, using a finer mesh requires using a smaller timestep to ensure stability of solution, consequently more timesteps are needed to be solved in order to simulate the same passage of time. In this way the type of solver also affects the scalability. Future implementation should use some more sophisticated time discretisation for example, a θ scheme.

A final aspect that needs to be solved is the application of boundary conditions. As mentioned in the section discussing the parallel implementation, this is currently a bug that must be carefully planned around when designing the mesh. The current code is not robust and although it cannot be considered a performance improvement, this problem must be resolved if the code is to be used in the future.

References

[1] A. Gerstenberger. *An XFEM Based Fixed-grid Approach to Fluid-structure Interaction.* 2010.

[2] R.W. Lewis, P. Nithiarasu, and K. Seetharamu. *Fundamentals of the Finite Element Method for Heat and Fluid Flow.* Wiley, 2004.

[3] J. N. Reddy. *An introduction to the finite element method.* McGraw-Hill, BostonBurr Ridge, ILNew York ... etc, 2006.