HIGH PERFORMANCE COMPUTING

Heat Conduction Equation

January 25, 2018

António Pedro Araújo Fraga

Student ID: 279654

Cranfield University

M.Sc. in Software Engineering for Technical Computing

Contents

Abstract

Three numerical schemes were applied to compute a solution for a parabolic partial differential equation, the heat conduction equation. It was used an explicit scheme and two implicit schemes. The solutions were computed both sequentially and in parallel making use of **MPI**, **M**essage **P**assing **I**nterface technology. Two pairs of **time** and **space** steps were studied, along with their effect on the solving system. **Speed-ups** of solutions computed with several processes were analysed, and their values were discussed.

Table 1: Nomenclature

Diffusivity	D
First derivative in time	$\frac{\partial f}{\partial t}$
First derivative in space	$\frac{\partial f}{\partial x}$
Time grid position	n
Space grid position	i
Function at time and space grid position	f_i^n
Time step	Δt
Space step	Δx
Time value	t
Space value	x
Analytical function at specific space and time values	$f(x,t)$
Initial Temperature	T_{in}
Surface Temperature	T_{sur}

Introduction

Numerical methods are used to obtain an approximated solution for a given problem. The exact solution of those problems are usually computed in **Nondeterministic Polynomial Time**, therefore an approximated solution is often accepted. The approximation factor is often related with the number of space steps that a time step is split on. A high number of space steps leads to a more accurate solution, keeping in mind the existence of **round-off** errors [1].

Although, decreasing the value of space steps results in a more "computational hungry" process. Which means that the time used to compute a solution increases, unless several processes are used to compute the final solution. Parallel computing is simple if it's done with **independent** data. In this case, there's no need to exchange data between workers. Sometimes the methods used to compute the solution have dependencies between space steps calculations. Thus, unless a process communicates with different processes, it's not possible to compute an approximated solution.

The challenge is to minimize the communication overhead. To achieve that, one has to

keep in mind how to perform communications. Deciding **when** and **how** to **exchange** data between processes.

Memory management plays an important role on **efficiency** as well. A program is split into **segments**, and each segment is split into **pages**. The size of each segment depends of the compiler. Each of them contains important data like the **stack**, **heap**, and actual data. But each **page** is divided into equal parts, **4kb** on UNIX based systems. Therefore, whenever a program accesses an address, a page is loaded into **cache**, which is a very fast type of memory close to the **CPU**. This means that accessing memory within the same page won't add the overhead of moving the data from the main memory to cache again. Such knowledge might make one think of how to write specific parts of the code.

Problem definition

The problem is defined in the previously developed **report**[1] for **Computational Methods** & **C++** modules. It is intended to examine the application of distributed memory parallel programming techniques on the referred problem.

Analysis

The analysis of **parallel programming** can be done by measuring times of execution. One should measure the time of execution with a single process, comparing it with the execution time in parallel with **p** processors. Therefore, the **Speedup** concept can be defined[2],

$$Speedup(P) = \frac{Time_{parallel}(P)}{Time_{Sequential}}$$

It is important to define the concept of **Theoretical Speedup** as well. Whenever a program is computed in parallel, the optimal speedup is obtained by taking the number of processes used to compute a solution. Often, the **Optimal Speedup** is difficult to be achieved. This happens because the solution is computed in a distributed system. Processes, by default, are not able to access data from different processes easily. One must use a form of **IPC**, or **I**nter **P**rocess **C**ommunication. The **MPI** technology offers an **API** to achieve that.

Procedures

Three different schemes/methods were used to compute a solution for the given problem, sequentially and in parallel. One of them is an explicit schemes, **Forward in Time, Central in Space**, and two of them are implicit schemes, **Laasonen Simple Implicit** and **Crank-Nicholson**. Two pairs **space** and **time** steps were studied,

- $\Delta t = 0.1$ and $\Delta x = 0.5$
- $\Delta t = 0.001$ and $\Delta x = 0.005$

It was seen in the previous work[1] that these schemes can be written in its discretized form.

A **time step** can be defined as an array, divided by the number of space steps to be computed. In a sequential algorithm, the process handles the entire domain. Since it was intended to parallelize the computational method, the **time step** can be divided into **p almost equal** parts, with **p** being the number of processes available. Notice that having **more** processes than space steps is an exception. Thereby, the **lower** and **upper** bounds of space steps to be computed by a process can be determined by it's ranking.

$$lower = \frac{ranking \times number_{SpaceSteps}}{number_{Processes}} \qquad upper = \frac{(ranking+1) \times number_{SpaceSteps}}{number_{Processes}} - 1$$

Defining these values as integers will avoid any kind of **floating point** values.

We can not, however, define these rules if it happens to have more available processes than space steps. In this case, only the number of processes corresponding to the number of space steps can be used.

Notice that the process with the **rank** = 0, computes the first window of values, whereas the process with **rank** = 1 computes the second window of values. The same idea is replicated to the next processes and windows.

Explicit Scheme

It is known that these type of schemes rely only on values from the previous time steps to compute the solution[1]. Therefore, once one has access to this values, it is possible to compute the values for the current time step. In this cases, processes that need values from different workers, can obtain them by using the MPI API.

Forward in Time Central in Space

The stencil for this scheme can be observed on **Figure 1**. When computing the **first** time step for this scheme, the previous time step values correspond to the **initial conditions**. This means that no data had to be exchanged during the calculations of that iteration. Every process had knowledge of these conditions.

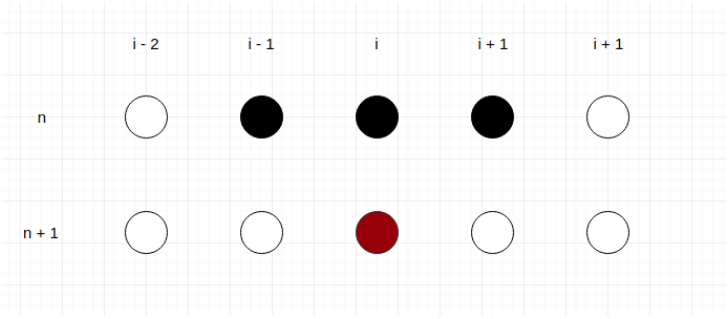

Figure 1: Forward in Time Central in Space method stencil.

However, the next time steps calculations of each process required values that didn't exist in their memory. **Non-blocking** communication was used to exchange dependable data. It was known that the next iteration to be computed by those processes needed the first time step value from the processes that were responsible to compute the next window of values. Therefore, whenever a process finished to compute the first value of an iteration, was responsible to send that value to the previous process. With the exception of the **root** process, that knows the value from the **Initial Conditions**. By this time, the processes could expect to receive a value from the next process, so the same **Non-blocking** communication was used to obtain it. **Figure 2** contains a graphic representation of the sending messages on the first iteration.

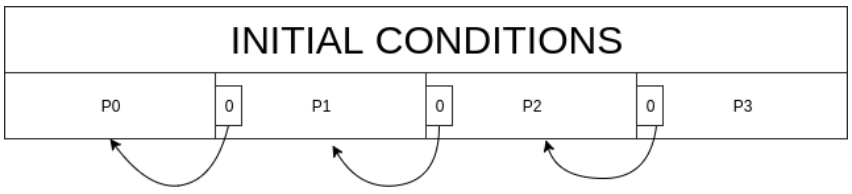

Figure 2: Processes sending first value to the previous worker.

Following the same mindset, whenever a process reached the last value to be computed, it was sending that information to the process responsible for the next window. Knowing that it would have to receive a value from the previous process. Notice that the **ranking** of the

receiving process could be discovered by incrementing its own ranking, and the ranking of the sending process was found by decrementing it. The process with the highest ranking was not sending or receiving any value from the next worker, because there was none. It could also access to the surface temperature value, a **known** and constant variable. The graphical representation of this phase can be observed on **Figure 3**.

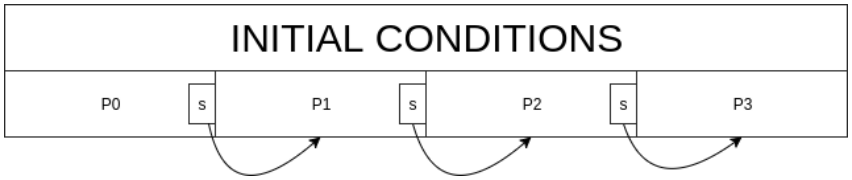

Figure 3: Processes sending first value to the previous worker.

The advantage of using **Non-blocking** communication, is that processes didn't have to go into the **waiting** state whenever they were sending or receiving messages. The **MPI Wait** call was used by the time processes were about to use those values. For the next iteration, whenever a process was computing the first and last value of its own window, it had to wait for the receiving value. The process would enter into a **waiting** state if the request was not fulfilled by that time.

This mechanism of communication was used in nearly every time step calculations, **except** for the **first** and **last** one, when there was no need of communications.

Implicit Schemes

Implicit schemes rely on both previous and current time step in order to calculate the desired value[1]. The methods stencil can be observed on **Figure 4** and **Figure 5**.

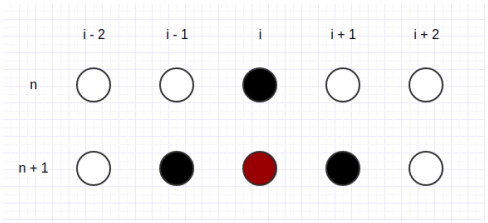

Figure 4: Laasonen's method stencil.

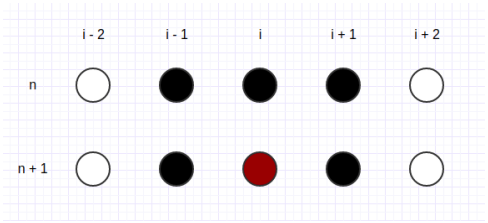

Figure 5: Crank-Nicholson's method stencil.

The time step computation of these schemes requires that a special form of a linear system of equations to be solved. This system $Ax = b$ contains a **tridiagonal matrix**, the

matrix **A**. In case of a sequential code, the **Thomas Algorithm** is the most efficient solution. With a parallel code, the **spike** algorithm can be used to solve dependencies among processes. The results can be achieved by splitting the computation methodology into three different phases.

- Solve **Ay = b** in parallel.
- Gather the top and bottom values of **y** into one process and solve dependencies, computing a vector **x**. Broadcast the solution.
- Solve **Sx = y** in parallel.

Since the matrix **A** is a **tridiagonal** matrix, it can be divided into several smaller matrices, a **block tridiagonal matrix**[3], like seen in **Figure 6**. Each process was responsible to solve a smaller linear system of equations, obtaining its own **y** array.

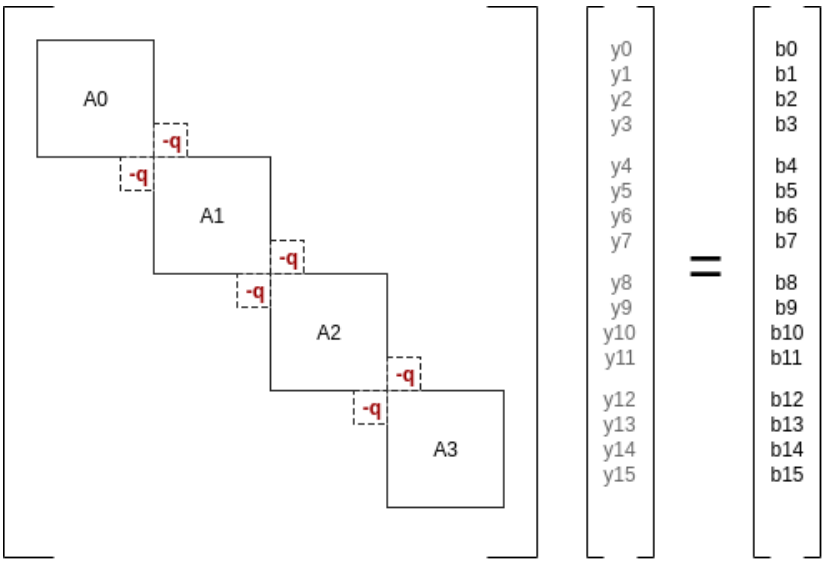

Figure 6: Division of main matrix in blocks.

This implies that several variables are not used in this phase (**-q**). Thus, two different arrays were created[4], **v'** and **w'**,

$$v' = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ -q \end{bmatrix} \qquad w' = \begin{bmatrix} -q \\ \cdots \\ 0 \\ 0 \end{bmatrix}$$

These arrays were used to compute the **spikes** arrays. They can be computed by solving **Av = v'** and **Aw = w'**, creating two different arrays, **v** and **w**. And since **A** is a tridiagonal matrix, they can be computed with **Thomas Algorithm** as well. These arrays are useful to create the matrix **S** of the last phase of the **spike** algorithm.

In order to solve the **Sx = y**, one has to solve a special form of this equation first. Involving every **top** and **bottom** element of the **y** array of each process, forming the array **y'**. This information was gathered by the **root** process. In order to solve dependencies between processes, the **S'x' = y'** system was solved. Note that the v_t, v_b, w_t and w_b values are the **top** and **bottom** values of both **v** and **w** arrays, the **spike arrays**, that were previously calculated.

$$\begin{bmatrix} 1 & 0 & v_t & & & & & & \\ 0 & 1 & v_b & & & & & & \\ & w_t & 1 & 0 & v_t & & & & \\ & w_b & 0 & 1 & v_b & & & & \\ & & & w_t & 1 & 0 & v_t & & \\ & & & w_b & 0 & 1 & v_b & & \\ & & & & & w_t & 1 & 0 \\ & & & & & w_b & 0 & 1 \end{bmatrix} \begin{bmatrix} x'0_t \\ x'0_b \\ x'1_t \\ x'1_b \\ x'2_t \\ x'2_b \\ x'3_t \\ x'3_b \end{bmatrix} = \begin{bmatrix} y'0_t \\ y'0_b \\ y'1_t \\ y'1_b \\ y'2_t \\ y'2_b \\ y'3_t \\ y'3_b \end{bmatrix}$$

This system of equations can be solved by using the **Gaussian Elimination** method, and the **x'**, can be *broadcasted* to the rest of the processes.

The last phase consists of solving the **Sx = y** system, that can be defined in **Figure 7**.

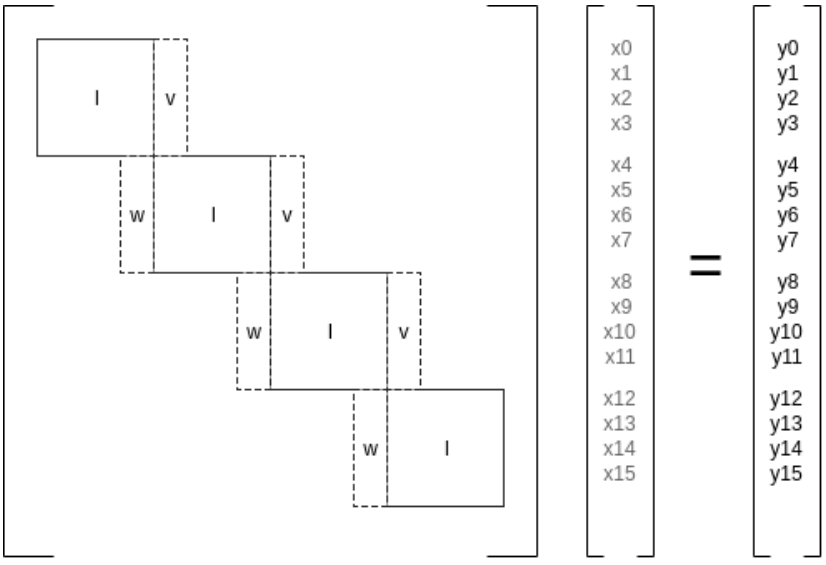

Figure 7: Sx = y system of equations.

Therefore, the solution can be obtained with[4],

$$\begin{cases} X_0 = Y_0 - VX_1^t \\ X_j = Y_j - VX_{j+1}^t - WX_{j-1}^b \\ X_{P-1} = Y_{P-1} - WX_{P-2}^b \end{cases}$$

Results & Discussion

The results of the four methods, **Richardson**, **DuFort-Frankel**, **Laasonen Simple Implicit** and **Crank-Nicholson** can be seen in the following figures/tables. These results were used to analyze each solution quantitatively and qualitatively. In most of the plot charts, the obtained solution was compared to the analytical solution so that it would be possible to realize whether the solution was a good approximation or not. Notice that the next results are regarding to the "default" values of time and space steps, $\Delta t = 0.01$ and $\Delta x = 0.05$.

Table 2: Richardson method error table.

t \ x	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
0.1	1.05136e+06	631856	123707	8417.73	300.81	8417.73	123707	631856	1.05136e+06
0.2	1.33245e+11	1.39e+11	7.02854e+10	2.06123e+10	6.93136e+09	2.06123e+10	7.02854e+10	1.39e+11	1.33245e+11
0.3	2.14659e+16	2.74969e+16	1.97012e+16	9.88337e+15	5.98161e+15	9.88337e+15	1.97012e+16	2.74969e+16	2.14659e+16
0.4	3.91917e+21	5.60267e+21	4.87086e+21	3.31281e+21	2.58429e+21	3.31281e+21	4.87086e+21	5.60267e+21	3.91917e+21
0.5	7.74272e+26	1.19047e+27	1.18021e+27	9.72231e+26	8.60626e+26	9.72231e+26	1.18021e+27	1.19047e+27	7.74272e+26

By examining **Table 2**, it could be concluded that the solution given by the Richardson method was considerably different from the analytical solution. This was due to the fact that this method is declared as **unconditionally unstable**. As referred before, when a method is declared unstable, the error grows as the time advances. The error growth was responsible for obtaining a different solution, or a solution to a different problem. The mathematical calculations regarding the stability and accuracy properties of this method can be found under the appendix section.

When looking at **Figure 7**, it can be observed that the DuFort-Frankel solution is quite approximated to the real solution. This scheme, as it could be observed at **Figure 11**, is

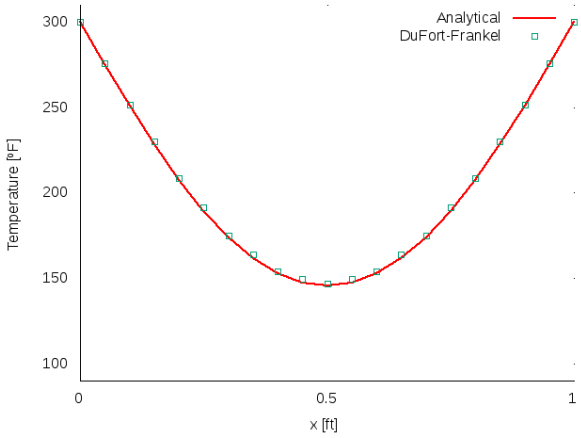

Figure 8: DuFort-Frankel's solution at $t = 0.5$.

more time efficient comparing to the implicit unconditionally stable methods, the only disadvantage is the fact that it requires a different method for the first iteration.

Similarly of what could be concluded on DuFort-Frankel results, by observing **Figure 9** and **Figure 8**, it can also be deducted that these are good solutions. These schemes, Crank-Nicholson and Laasonen, are unconditionally stable as well. Therefore good results were expected.

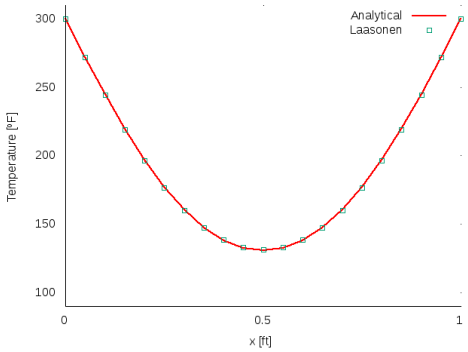

Figure 9: Laasonen's solution at $t = 0.4$.

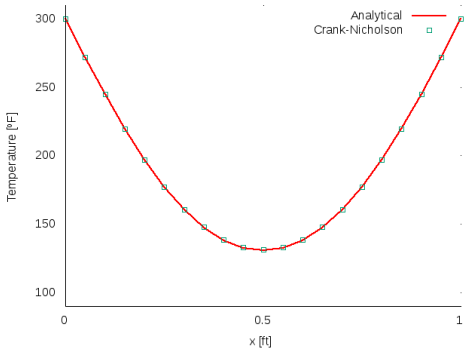

Figure 10: Crank-Nicholson's solution at $t = 0.4$.

In other hand, when a quantitative analysis was done, it could be seen that the Crank Nicholson scheme is more accurate than the Laasonen and DuFort-Frankel methods. By looking at **Figure 10**, it can be observed that the second norm value of the **Error matrix** of this scheme is smaller than the values obtained by the other methods **Error Matrices**.

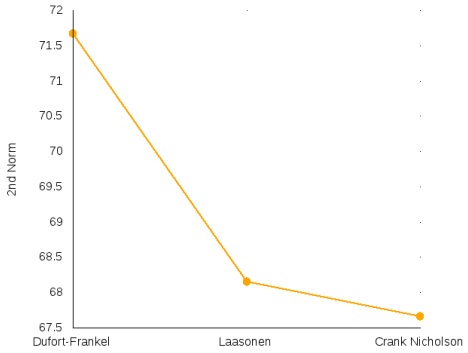

Figure 11: 2nd norm values of Error Matrices.

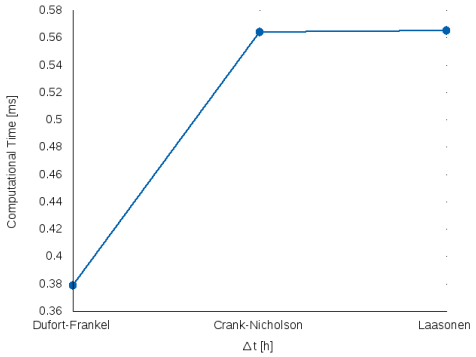

Figure 12: Computational times of stable methods.

Laasonen Implicit Scheme: study of time step variation

Laasonen Implicit Scheme is an unconditionally stable scheme to solve Parabolic Partial Differential Equations. Therefore, with the right time and space step, there's almost no error related to the development of its results throughout the time advancement.

A reduction on these steps led to a higher computational time, since there's more calculations to be made. Whereas steps with higher values led to more inaccurate results[**?**]. This phenomenon could be explained with a concept that was introduced earlier, the **truncation error**[**?**]. This error can only be avoided with exact calculations, but can be reduced by applying a larger number of smaller intervals or steps. As referred before, different results of this method were studied by changing the time step size. The space step was maintained, $\Delta x = 0.05$.

Table 3: Laasonen method error table for the several Δt at $t = 0.5$

x / Δt	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
0.01	0.288694	0.385764	0.255427	0.0405061	-0.0611721	0.0405061	0.255427	0.385764	0.288694
0.025	0.738044	1.0344	0.805442	0.368491	0.157551	0.368491	0.805442	1.0344	0.738044
0.05	1.53627	2.15669	1.71375	0.864487	0.457364	0.864487	1.71375	2.15669	1.53627
0.1	3.29955	4.49523	3.46045	1.7082	0.898726	1.7082	3.46045	4.49523	3.29955

Table 3 and **figure 12** could support the previous affirmations. While observing **table 3**, it could be seen that the error is larger for bigger time steps, as it was expected. Whereas when observing **figure 12**, it can be identified a reduction in computational time as the **time step** becomes larger.

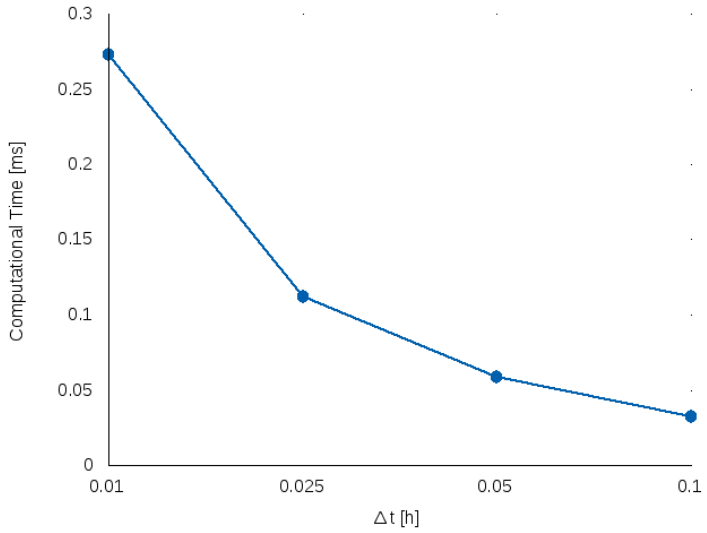

Figure 13: Laasonen method computational times for the several Δt.

Conclusions

The obtained results could support the theoretical concepts. Unstable methods demonstrated an error growth through the time progress. The **Forward in Time, Central in Space** explicit scheme was stable with the given initial conditions, therefore it could support a good solution for the explicit stable scheme, **DuFort-Frankel**. As referred, the solution of the DuFort-Frankel method strongly depends on the first iteration solution.

It could be observed that smaller steps can lead to a time expensive solution, whereas larger steps lead to an error increase. Stable methods could give a good solution with the right time and space steps, but by analysing the second norm value, it was concluded that the Crank-Nicholson method is more accurate. This is due to the fact that this method has a better approximation order.

It is important to have a balance between the two problems (time and approximation), a method should be computed in an acceptable time, and still obtain a good result. In realistic scenarios the problem solution is not known, therefore error estimates are impractical. The used step size should be small as possible, as long as the solution is not dominated with round-off errors. The solution must be obtained with a number of steps that one has time to compute.

References

[1] António Pedro Fraga, December 2017, *Heat Conduction Equation, C++ & Computational Methods*, Available at: <http://pedrofraga.me/heat-conduction-equation.pdf> [Accessed 24 January 2018]

[2] Multiple authors, May 2017, *A Comprehensive Linear Speedup Analysis for Asynchronous Stochastic Parallel Optimization from Zeroth-Order to First-Order*, Available at: <http://xrlian.com/res/Asyn_Comprehensive/paper.pdf> [Accessed 24 January 2018]

[3] Li-Wen Chang, 2014, *Scalable Parallel Tridiagonal Algorithms with Diagonal Pivoting and their optimization for Many-Core Architectures*, Available at: <http://impact.crhc.illinois.edu/shared/Papers/Chang-Thesis-2014.pdf> [Accessed 25 January 2018]

[4] Eric Polizzi, Ahmed H. Sameh, February 2016, *A parallel hybrid banded system solver: the SPIKE algorithm*, Available at: <http://impact.crhc.illinois.edu/shared/Papers/Chang-Thesis-2014.pdf> [Accessed 25 January 2018]

Appendices