High Performance Computing

Heat Conduction Equation

January 26, 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],

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

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

Every solution was computed on **Delta**, a supercomputer located at Cranfield University. The computational time was measured by using the **MPI Wtime** call, and assuring that every process was entering and leaving the computation of a given method at the same time.

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.

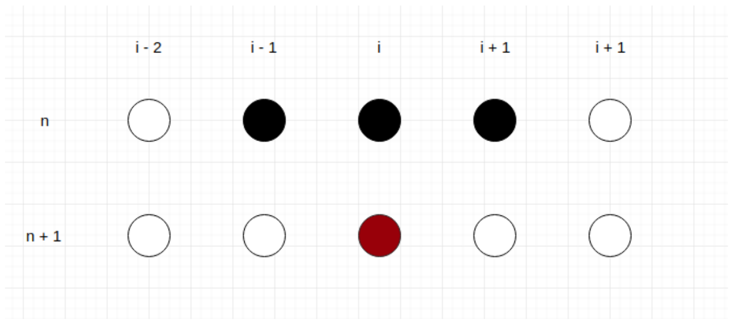

Figure 1: Forward in Time Central in Space method stencil.

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.

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

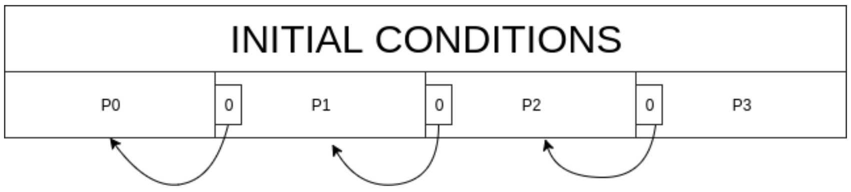

Figure 2: Processes sending first value to the previous worker.

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**.

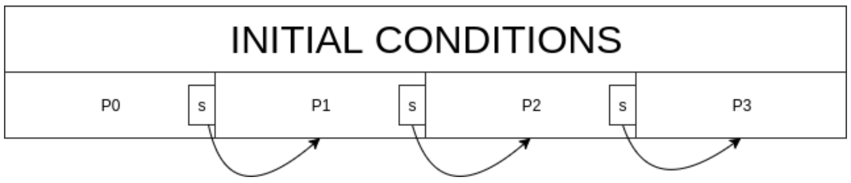

Figure 3: Processes sending first value to the previous worker.

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**.

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

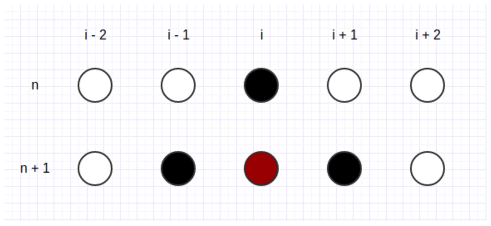

Figure 4: Laasonen's method stencil.

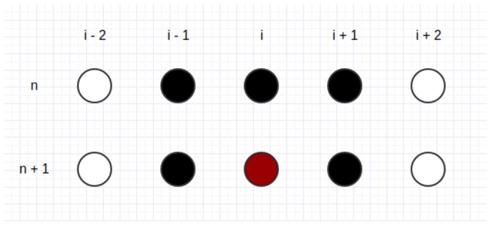

Figure 5: Crank-Nicholson's method stencil.

matrix **E**, which can be divided into two different matrices, **A** and **S**[4]. 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.

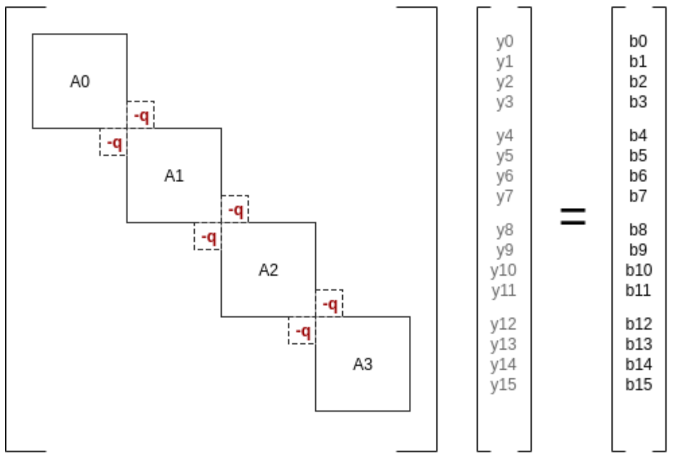

Figure 6: Division of main matrix in blocks.

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

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

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**.

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

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

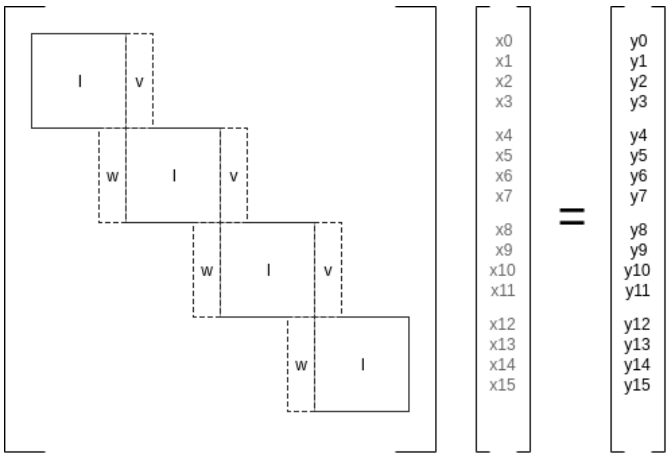

Figure 7: Sx = y system of equations.

Results & Discussion

In this section the results of the experiments were analysed. Starting by observing **Figure 8**, it can be seen that the solution of the **explicit** scheme is not accurate at these conditions. Notice that this solution was computed by **4** processes, with a $\Delta t = 0.001$ and $\Delta x = 0.005$. This phenomenon can be explained by making the stability analysis of this method[1]. On this conditions, this method is unstable, therefore it can not present an approximated solution. It can be seen that the implicit schemes, however, were able to present good results.

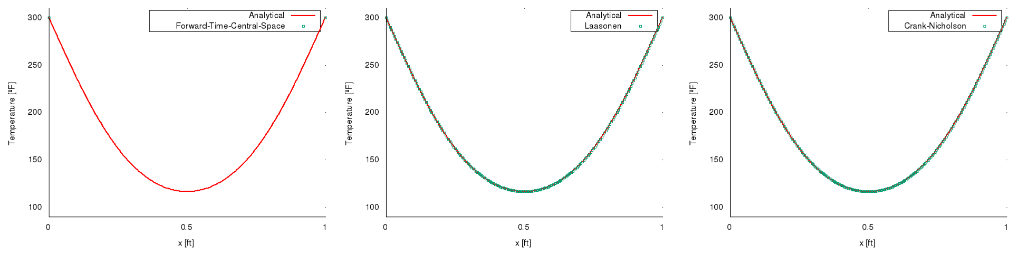

Figure 8: FTCS, Laasonen and Crank-Nicholson solutions calculated with 4 processors, at t = 0.4, with $\Delta t = 0.001$ and $\Delta x = 0.005$.

As referred, it was possible to measure the computation time of a given method. The results for a sequential calculations were quite similar to the previous results[1]. The execution time of a solution for $\Delta t = 0.001$ **and** $\Delta x = 0.005$ did was stable as the number of processes were increased. This happened because the **software** implementation allowed only one process to work. Notice that under these conditions, each time step contains **three** different values. **Two** of those values are known, as they are considered as **initial conditions**.

Therefore, there was only one value to be computed for every time step.

The results of a parallel execution for the $\Delta t = 0.001$ **and** $\Delta x = 0.005$ can be observed on **Figure 9**. Under these conditions, **199** values had to be computed for every time step. By observing the figure, and comparing the sequential solution execution time of the methods, we can see a continuous improvement of the **Forward in Time, Central in Space** scheme. This improvement couldn't be replicated by the implicit schemes. The execution time of these schemes started to increase when the solution was calculated by more than **four** processes.

It was referred that the dependencies among the several processes computing the **Forward in Time, Central in Space** solution were handled using **non-blocking** communication. Whereas the implicit schemes were using a blocking communication to solve their dependencies. A blocking communication was used because the **second** phase of the **spike** algorithm required a sequential flow of data. Thereby, one can conclude that this phase was a major bottleneck while measuring the execution time of these solutions.

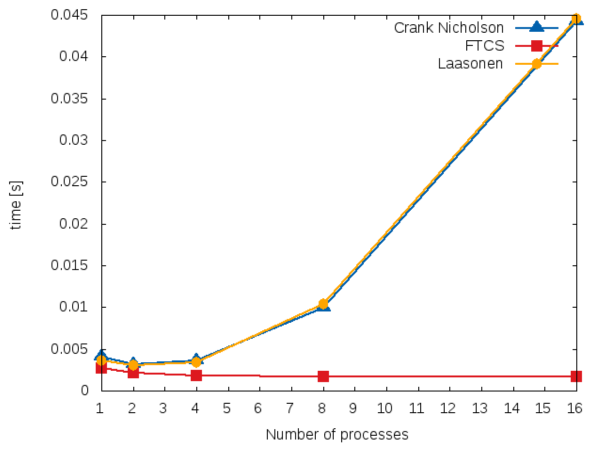

Figure 9: Comparison of execution times.

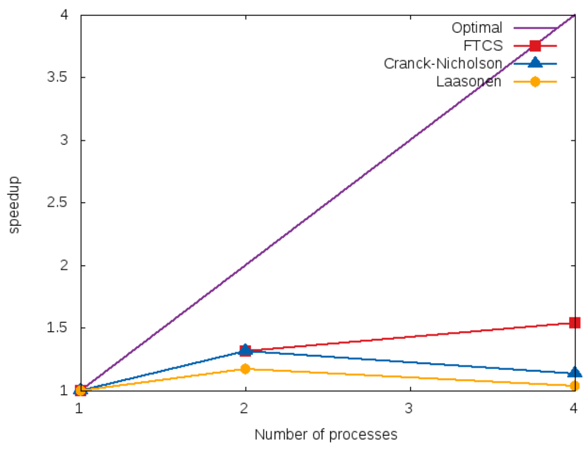

Figure 10: Speedup of each method.

However, while observing **Figure 10**, it could be seen that the **speedup** value of the **Forward in Time, Central in Space** scheme is far from its optimal value. In fact this value is significantly low. A larger **speedup** value would be seen while computing a larger grid. In such small system, the bottleneck of communications is high, and therefore one can not obtain big improvements. An experiment with a system of **thousands** of elements in each time step would contribute to a later perceived bottleneck. Clarifying, the bottleneck would only be visible with a higher number of processes.

The implemented **tridiagonal** solver was compared to the **Lapack** library **tridiagonal** solver. While observing **Figure 11**, it can be seen that the execution time of the implemented solver was slower than the Lapack one. The **Thomas algorithm** has a **linear** time complexity,

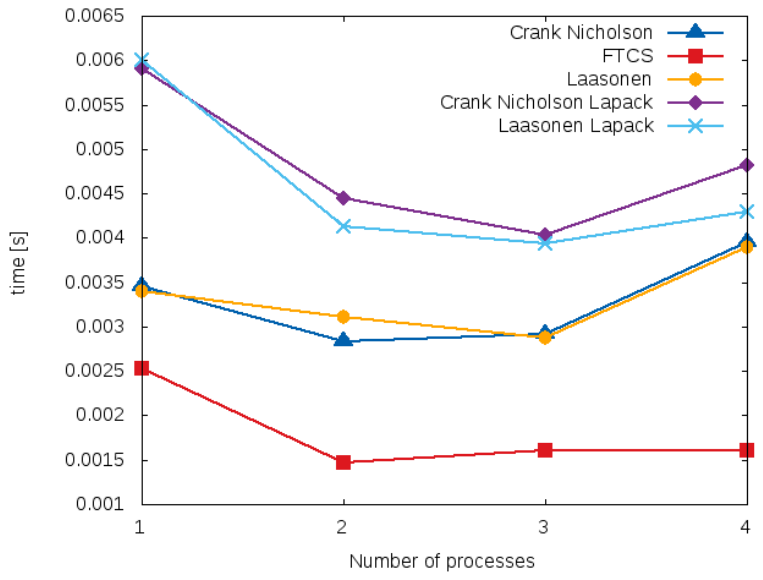

Figure 11: Comparison of execution times with one process.

making it one of the fastest algorithms in sequential code. The significantly slow time obtained by the Lapack implementation may have to do with the architecture where the code is running. It can be possibly explained with **memory management** matters as well. It's also possible that these methods would behave better with a larger grid.

Conclusions

In conclusion, computational power may not be the more important factor when dealing with High Performance Computing.

It was seen that memory plays an important role on this area as well. In order to build an efficient solver, one must take attention of **how** to deal with the several **data structures** used to obtain a solution.

The overhead of communicating among processes is often a major bottleneck that one must minimize as much as possible. Is also known that a solution with independent data can be more efficient than systems which rely on information spread across the solution. Whenever the number of communications between processes is significantly high, it is expected to obtain less efficient solvers.

The use of distributed systems becomes advantageous when dealing with large problems. Problems that often require a high number of calculations within a process. Those calculations can often be distributed by workers in order to minimize execution times. Bigger solution tend to minimize the communication bottleneck as well. Whenever the ratio between number of communications and work load per process is minimized, one can expect a better performance.

Like referred before, building a system requires a "near to optimal" level of synchronization. It is implied to think **when**, **how** and **what** to exchange between workers.

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]