Runtime comparison between Chapel and Fortran

Willian Carlos Lesinhovski
wlesin@yahoo.com.br
Department of Environmental Engineering, Federal University of Paraná
Curitiba, PR, Brazil

Nelson Luís Dias
nelsonluisdias@gmail.com
Department of Environmental Engineering, Federal University of Paraná
Curitiba, PR, Brazil

Livia Souza Freire
liviafreire@usp.br
University of São Paulo
São Carlos, SP, Brazil

Anna Caroline Felix Santos de Jesus
carolinefelix@usp.br
University of São Paulo
São Carlos, SP, Brazil

ABSTRACT

In this text we present a simple but interesting runtime comparison between Chapel and Fortran when performing some very common algorithms in numerical analysis: matrix multiplication, Lax method for the kinematic wave equation and SOR method for the Poisson equation. Chapel presented a very satisfactory performance reducing the processing time from 10% to 50% compared to Fortran.

KEYWORDS

Chapel vs Fortran, desktop computing, numerical methods

ACM Reference Format:
Willian Carlos Lesinhovski, Nelson Luís Dias, Livia Souza Freire, and Anna Caroline Felix Santos de Jesus. 2023. Runtime comparison between Chapel and Fortran. In *Proceedings of The 10th Annual Chapel Implementers and Users Workshop (CHIUW 2023)*. ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Numerical simulations in fluid mechanics demand a high computational cost, often requiring the use of supercomputers and parallel programming. In general, models are programmed in Fortran because of its speed. In view of that, since we want to use Chapel for fluid mechanic simulations, we decided to perform some runtime comparisons between these two programming languages to ensure that Chapel is in fact a good choice for our research purposes.

The numerical methods used in fluid mechanics in general are divided into several subroutines. Therefore, we chose some of the most relevant ones for our purposes to compare the performance of both languages in each one of them. In this way, it is easier to analyze the strengths and weaknesses of each language while ensuring that the algorithms are coded in essentially the same way.

For the comparisons we chose: multiplication of matrix by vector; solution of the kinematic wave equation by the Lax method and solution of the Poisson equation by the SOR method. Although these algorithms can be programmed in parallel, our focus here is to compare the performance of the code generated by each language on a single core. Related work is also being reported on the efficiency of parallelized Chapel × Fortran on a single locale, see [1].

Tests were performed on a computer with a Intel Core i9-12900K processor and a Linux Mint 21.1 operating system. For compilation we use the flag `-O3` in Fortran and `--fast` in Chapel 1.29.

CHIUW 2023, June 1-2 2022, Online Conference
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 MATRIX VECTOR MULTIPLICATION

Let x, y be real vectors of size $n \in \mathbb{N}$ and A a real matrix of size $n \times n$ with elements a_{ij}. The product $y = Ax$ is defined by

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \; i \in \{1, \ldots, n\}. \tag{1}$$

Matrix-vector multiplication is used in many algorithms, so a well-optimized routine to do it is of great help and interest. In general, the runtime of the Ax product can be improved using low-level routines and advanced matrix multiplication algorithms [2]. For Fortran there is the `matmul` function and in Chapel we can use the `gemv` function from the BLAS library. However, for comparison purposes we will also implement the product in a standard way.

The algorithm for calculating (1) in Chapel is as follows

```
for i in 1..n do {
   var sum = 0.0;
   for j in 1..n do {
      sum += A[i,j]*x[j];
   }
   y[i] = sum;
}
```

As the calculation of y_i uses only elements from the row i of A, programming languages that store arrays considering row-major order (Chapel) have an advantage over those that use column-major order (Fortran). On the other hand, if we define the multiplication $y = x^T A$ as

$$y_i = \sum_{j=1}^{n} x_j a_{ji}, \; i \in \{1, \ldots, n\}, \tag{2}$$

only the elements of column i of A are used for the calculation of y_i and languages that use column-major order have an advantage. In this case the algorithm in Fortran becomes

```
do i = 1,n
```

```
   sum = 0.0
   do j = 1,n
      sum = sum + x(j)*A(j,i)
   end do
   y(i) = sum;
end do
```

Table 1 shows the time in seconds to calculate Ax and $x^T A$ using a standard algorithm to compute (1) and (2) and also Ax with the aforementioned low-level functions in the two languages. In the tests we chose $n = 10000$ and A and x were filled with random values.

Table 1: Runtime of matrix vector multiplication.

Language	Ax	$x^T A$	Ax (gemv/matmul)
Chapel	0.0820	0.5541	0.0278
Fortran	0.3625	0.0523	0.0340

The standard algorithm for Ax is faster in Chapel than in Fortran, but for $x^T A$ we have the opposite: $x^T A$ in Fortran is faster than Ax in Chapel. Thus, we can see that the way of storing the matrix and how its elements are accessed has a large influence on the performance of the codes. As expected, the low-level functions provide a considerable speed gain with gemv being the fastest option among those tested.

3 KINEMATIC WAVE EQUATION

Lax's method for the kinematic wave equation is a fairly simple method and is a good starting point to compare Chapel and Fortran when solving differential equations. Consider the kinematic wave equation given by

$$\frac{\partial}{\partial t}u(x,t) + c\frac{\partial}{\partial x}u(x,t) = 0, \tag{3}$$

with domain $x \in [0, 10]$, $t \in [0, 1]$ and the following initial conditions

$$u(x,0) = \begin{cases} 2x(1-x), & \text{if } 0 \leqslant x \leqslant 1, \\ 0, & \text{if } 1 < x \leqslant 10, \end{cases}$$
$$u(0,t) = u(10,t) = 0,\ 0 \leqslant t \leqslant 1.$$

For the discretization of the equation let us consider the grid (x_i, t_n) defined by $x_i = i\Delta x$ where $i \in \{0, 1, \ldots N_x\}$ with $\Delta x = 10/N_x$ and $t_n = n\Delta t$ where $n \in \{0, 1, \ldots N_t\}$ with $\Delta t = 1/N_t$. The approximate solution $u_i^n \approx u(i\Delta x, n\Delta t)$ calculated using the Lax-Friedrichs method is given by the relation:

$$u_i^{n+1} = \frac{1}{2}\left[u_{i+1}^n + u_{i-1}^n - \sigma(u_{i+1}^n - u_{i-1}^n)\right],$$

where

$$\sigma = \frac{c\Delta t}{\Delta x}.$$

There are two options for storing the values of u in an array: saving the approximations of each instant of time in a row or in a column. The first one is more efficient in programming languages that use row-major-order while the second one is better for programming languages that use column-major-order. We will test both in each language. Also, since in Lax's method only the information from the time t_n is used to calculate the values in t_{n+1}, it is possible to write the code in such a way that only the values of these two instants of time are stored in the array replacing u_i^{n-1} by u_i^{n+1} for $n \geqslant 1$.

The code of the first option in Chapel is

```
var nold = 0;
var nnew = 1;
for n in 1..Nt do  {
   for i in 1..Nx-1 do {
      u[nnew,i] = 0.5*((u[nold,i+1] + u[nold,i-1])-
       cour*(u[nold,i+1]-u[nold,i-1]));
   }
   u[nnew,0] = 0.0;
   u[nnew,Nx] = 0.0;
   nnew <=> nold;
}
```

and the code of the second option in Fortran is

```
nold = 0
nnew = 1
do n = 1,Nt
   do j = 1,Nx-1
      u(j,nnew)=0.5*((u(j+1,nold)+u(j-1,nold))-&
       cour*(u(j+1,nold)-u(j-1,nold)))
   end do
   u(0,nnew) = 0.0
   u(Nx,nnew) = 0.0
   nk = nnew
   nnew = nold
   nold = nk
end do
```

For the tests we set $N_x = 20000$, $N_t = 10000$ and $c = 2$. Table 2 shows the runtime in seconds of the codes in Chapel and Fortran. Comparing the best results for each language, Chapel performed significantly better than Fortran reducing the runtime in half. Furthermore, in both languages the codes that use the best option to build the array are significantly faster than those that use the worst option.

Table 2: Runtime of Lax method in Chapel

Language	Rows	Columns
Chapel	0.0971	0.2286
Fortran	0.3492	0.1893

4 POISSON EQUATION

In many numerical methods for solving the Navier-Stokes equations it is necessary to solve a Poisson equation for the pressure term at each time step. Therefore, comparing the numerical solution of the Poisson equation, typically done with the SOR method, is of great relevance for our research purposes.

Consider the Poisson equation given by

$$\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f, \tag{4}$$

with domain $D = [0, 1] \times [0, 1]$ where

$$f(x,y) = -(\pi^2)(x^2 + y^2)\sin(\pi xy),$$

with the following boundary conditions

$$\begin{cases} u(x,1) = \sin(\pi x), \\ u(1,y) = \sin(\pi y), \\ u(x,0) = u(0,y) = 0. \end{cases}$$

For the discretization of the equation let us consider the grid (x_i, y_j) defined by $x_i = i\Delta l$ and $y_j = j\Delta l$ where $i,j \in \{0,1,\ldots N\}$ with $\Delta l = 1/N$. Considering a central finite difference scheme for the second order derivatives and applying the SOR method with relaxation parameter ω we have the following iterative algorithm to solve the Poisson equation

$$\begin{cases} \delta u^k_{i,j} = \omega\left((u^k_{i+1,j} + u^{k+1}_{i-1,j} + u^k_{i,j+1} + u^{k+1}_{i,j-1} - \Delta l^2 f_{i,j})/4 - u^k_{i,j}\right), \\ u^{k+1}_{i,j} = u^k_{i,j} + \delta u^k_{i,j}. \end{cases}$$

The algorithm stops when the difference of two consecutive iterations is less than a given tolerance ϵ, that is: when for some k we have

$$\frac{1}{(N-1)^2}\sum_{i,j=1}^{N-1} |\delta u^k_{i,j}| < \epsilon.$$

Based on the results of previous tests, we already know that the way to build the arrays and access their elements has a huge impact on the processing time of the algorithms. Therefore, for the SOR method we will only consider the best option for each programming language.

Due to the symmetry of the problem, it is only necessary to change the order in which the elements are updated. The code for the SOR algorithm in Chapel is

```
var err = 2.0*epsilon;
var k = 0;
while err >= epsilon do {
   err = 0.0;
   for i in 1..N-1 do {
      for j in 1..N-1 do {
         var um = (u[i+1,j]+u[i-1,j]+u[i,j-1]+
            u[i,j+1]-h2*f[i,j])/4.0;
         var du = omega*(um - u[i,j]);
         u[i,j] += du;
         err += abs(du);
      }
   }
   k += 1;
   err /= N2;
}
```

and the code for the SOR algorithm in Fortran is

```
error = 2*eps
k = 0
do while (error >= eps)
   error = 0.0
   do j = 1,N-1
      do i = 1,N-1
         um = (u(i+1,j)+u(i-1,j)+u(i,j-1)+&
            u(i,j+1)-h2*f(i,j))/4.0
         du = omega*(um - u(i,j))
         u(i,j) = u(i,j) + du
         error = error + abs(du)
      end do
   end do
   k = k+1
   error = error/N2
end do
```

Table 3 shows the runtime in seconds and the number of iterations of the SOR Method for $\omega = 1.95$, $N = 512$ and $\epsilon = 10^{-8}$ with initial guess $u^0_{i,j} = 0$ in the internal points of the grid. We see that after the same number of iterations the method was about 10% faster in Chapel than in Fortran.

Table 3: Runtime of SOR method

Language	Runtime	Iterations
Chapel	7.4721	7507
Fortran	8.3910	7507

5 CONCLUSIONS

The algorithms presented in this text are straightforward, with arrays and loops done serially. As a result, the codes in Chapel are very similar to those in Fortran allowing a direct comparison of performance between the two languages which is our goal. Furthermore, Chapel has some interesting features and advantages over Fortran. For example, swapping values between two variables in Chapel is done with one line of code using the command <=>, on the other hand in Fortran three lines of code and an auxiliary variable are required. Also, in Fortran it is necessary to declare all the loop variables, which is not necessary in Chapel.

Based on the results obtained in this work, even not performing tests with parallel algorithms, we can conclude that Chapel can be somewhat faster than Fortran. Therefore, we decided to use Chapel for the implementation of our fluid mechanics model due to its competitive performance compared to Fortran. Also, our target programs will require parallel processing which is much easier to do in Chapel than in Fortran.

6 ACKNOWLEDGMENTS

This study was funded by Brazil's Coordination for the Improvement of Higher Education Personnel (CAPES).

REFERENCES

[1] Anna de Jesus et al. 2023. Parallel Implementation in Chapel for the Numerical Solution of the 3D Poisson Problem. In *Chapel Implementers and Users Workshop*.

[2] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13 (1969), 354–356.