Runtime comparison between Chapel and Fortran

Willian. C. Lesinhovski\textsuperscript{1} Nelson. L. Dias\textsuperscript{1} Livia. S. Freire\textsuperscript{2} Anna. C. F. S. de Jesus\textsuperscript{2}

\textsuperscript{1}Department of Environmental Engineering
Federal University of Paraná

\textsuperscript{2}Instituto de Ciências Matemáticas e de Computação
University of São Paulo

June 2023
**Objective**

Compare the performance of Chapel and Fortran on a single core when running some classic algorithms in numerical analysis.

- Matrix vector multiplication;
- Lax-Friedrichs method for kinematic wave equation;
- SOR method for Poisson equation.
Objective

Compare the performance of Chapel and Fortran on a single core when running some classic algorithms in numerical analysis.

- Matrix vector multiplication;
- Lax-Friedrichs method for kinematic wave equation;
- SOR method for Poisson equation.

Motivation

Develop a code in Chapel for fluid mechanic simulations.
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

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \ i \in \{1, \ldots, n\}. \quad (1)$$

On the other hand, $y = x^T A$ is defined as

$$y_i = \sum_{j=1}^{n} x_j a_{ji}, \ i \in \{1, \ldots, n\}, \quad (2)$$

- $y = Ax$ is more efficient in programming languages that store arrays considering row-major order (Chapel).
- $y = x^T A$ is more efficient in programming languages that use column-major order (Fortran).

The runtime of the $Ax$ product can be improved using low-level routines and advanced matrix multiplication algorithms.
Chapel ($y = Ax$)

```chapel
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;
}
```

Fortran ($y = x^T A$)

```fortran
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
```

Low level functions for $y = Ax$

- `gemv` (Chapel)
- `matmul` (Fortran)
Results for matrix vector multiplication

- \( A \) is a real \( n \times n \) matrix with \( n = 10000 \);
- \( A \) and \( x \) were filled with random values.
Results for matrix vector multiplication

- $A$ is a real $n \times n$ matrix with $n = 10000$;
- $A$ and $x$ were filled with random values.

<table>
<thead>
<tr>
<th>Language</th>
<th>$Ax$</th>
<th>$x^TA$</th>
<th>$Ax$ (gemv/matmul)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td>0.0820</td>
<td>0.5541</td>
<td>0.0278</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.3625</td>
<td>0.0523</td>
<td>0.0340</td>
</tr>
</tbody>
</table>

Table: Runtime of matrix vector multiplication.
Kinematic wave equation

\[
\frac{\partial}{\partial t} u(x, t) + c \frac{\partial}{\partial x} u(x, t) = 0,
\]

with domain \( x \in [0, 10], t \in [0, 1] \).

The grid \((x_i, t_n)\) is defined by

- \( x_i = i \Delta x \) where \( i \in \{0, 1, \ldots N_x\} \) with \( \Delta x = 10/N_x \);
- \( t_n = n \Delta t \) where \( n \in \{0, 1, \ldots N_t\} \) with \( \Delta t = 1/N_t \).

The approximate solution \( u_i^n \approx u(i\Delta x, n\Delta t) \) is calculated using 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

\[
 \sigma = \frac{c \Delta t}{\Delta x}.
\]
Chapel

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;
}

Fortran

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
Results for kinematic wave equation

Boundary conditions

\[ u(x, 0) = \begin{cases} 
2x(1 - x), & \text{if } 0 \leq x \leq 1, \\
0, & \text{if } 1 < x \leq 10,
\end{cases} \]

\[ u(0, t) = u(10, t) = 0, \quad 0 \leq t \leq 1. \]

Parameters: \(N_x = 20000\), \(N_t = 10000\) and \(c = 2\).
Boundary conditions

\[ u(x, 0) = \begin{cases} 
2x(1 - x), & \text{if } 0 \leq x \leq 1, \\
0, & \text{if } 1 < x \leq 10,
\end{cases} \]

\[ u(0, t) = u(10, t) = 0, \quad 0 \leq t \leq 1. \]

Parameters: \( N_x = 20000, \) \( N_t = 10000 \) and \( c = 2. \)

<table>
<thead>
<tr>
<th>Language</th>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td>0.0971</td>
<td>0.2286</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.3492</td>
<td>0.1893</td>
</tr>
</tbody>
</table>

Table: Runtime of Lax method in Chapel
The Poisson equation

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

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

The grid $(x_i, y_j)$ is 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 parameter $\omega$ we have the following iterative algorithm to solve the Poisson equation

$$
\begin{align*}
\delta u_{i,j}^k &= \omega \left( (u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - \Delta l^2 f_{i,j})/4 - u_{i,j}^k \right), \\
u_{i,j}^{k+1} &= u_{i,j}^k + \delta u_{i,j}^k.
\end{align*}
$$

The stopping criteria is:

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

```chapel
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;
}
```

Fortran

```fortran
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
```
Results for SOR method

Source term

\[ f(x, y) = -(\pi^2)(x^2 + y^2) \sin(\pi xy). \]

Boundary conditions

\[
\begin{align*}
  u(x, 1) &= \sin(\pi x), \\
  u(1, y) &= \sin(\pi y), \\
  u(x, 0) &= u(0, y) = 0.
\end{align*}
\]

Parameters: \( N = 512, \omega = 1.95 \) and \( \epsilon = 10^{-8} \).
Initial guess in the internal points: \( u_{i,j}^0 = 0 \).
Results for SOR method

Source term

\[ f(x, y) = -(\pi^2)(x^2 + y^2) \sin(\pi xy). \]

Boundary conditions

\[
\begin{align*}
    u(x, 1) &= \sin(\pi x), \\
    u(1, y) &= \sin(\pi y), \\
    u(x, 0) &= u(0, y) = 0.
\end{align*}
\]

Parameters: \( N = 512, \omega = 1.95 \) and \( \epsilon = 10^{-8} \).

Initial guess in the internal points: \( u^0_{i,j} = 0 \).

<table>
<thead>
<tr>
<th>Language</th>
<th>Runtime</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td>7.4721</td>
<td>7507</td>
</tr>
<tr>
<td>Fortran</td>
<td>8.3910</td>
<td>7507</td>
</tr>
</tbody>
</table>

Table: Runtime of SOR method
Conclusions

- The codes in Chapel are very similar to those in Fortran allowing a direct comparison of performance between the two languages.
- Chapel can be somewhat faster than Fortran in a single core.
- We decided to use Chapel for the implementation of our fluid mechanics model due to its competitive performance compared to Fortran.
- Our target programs will require parallel processing which is much easier to do in Chapel than in Fortran.
- Chapel has some interesting features and advantages over Fortran.
  - 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.
  - In Chapel is not necessary to declare the loop variables.
Thank you for your attention!