ISSN: 2756-2638
Transactions on Engineering and Computing Sciences; Vol. 14 No. 02 (2026) (01-14)
https://doi.org/10.14738/tmlai.1402.20134

Array Access and Performance Regarding Numerical Algorithms

Arjen Markus[1] & Lenore Mullin[2]

1. Deltares Research Institute, The Netherlands
2. University at Albany, SUNY, USA

Abstract: Arrays are an important data structure in the solution of many numerical problems. In this study, the classical Poisson equation in two dimensions is used to gain insight in the consequences of the actual implementation and various possible compiler options for the performance: what is the optimal (fastest) solution? The equation is solved using a straightforward algorithm for a wide range of grid sizes to examine the effect of the memory management. The implementations include explicit loops and array operations, as well as parallellisation via OpenMP. Some general conclusions are drawn with respect to achieving good or optimal performance.

Keywords: Programming, Array operations, Mathematics, Fortran.

INTRODUCTION

We study how array access affects performance and try to identify how various forms of "dimension lifting" can improve performance. Dimension lifting is defined to be a way of splitting loops to map over various levels of memory, e.g. vector registers, prefetching mechanisms, shared memory multi-processors, etc [1,2]. We also study Fortran since it is used to implement many important scientific algorithms, e.g. the Poisson equation. Our goal is also to identify how to design and implement this algorithm given the many compilers available with reproducible, optimal performance. Moreover, we hope that these implementations can be achieved with minimal user effort.

Ideally, expressing the algorithm in a high-level formalism, that is, express what we want done, rather than how it should be done, would enable the compiler to decide on the best way to achieve this [3,4]. Modern Fortran does allow the programmer to hide many details behind features as array operations [5], but the resulting performance may not be optimal.

DESCRIPTION OF THE MATHEMATICAL PROBLEM

The Poisson equation can be encountered in a wide variety of physical problems, for instance as a description of the electrical potential due to charges distributed over space or the distribution of the temperature in a solid.[*]

Mathematically speaking, the equation in two dimensions reads:

[*] See for instance https://en.wikipedia.org/wiki/Poisson's_equation

$$\frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} = f(x,y) \tag{1}$$

where Φ is the physical quantity that is being studied and f(x,y) a forcing function.

The area over which this equation has to be solved can be of any shape, but often it is taken to be a rectangle. The boundaries of the rectangle can be open, allowing a *flux* of the quantity to enter or leave the area (a particular value for the quantity Φ or the flux is prescribed), or closed, in which case there is no flux, which translates into a boundary condition of the form $\partial\Phi/\partial\underline{n} = 0$, $\underline{n}$ being the normal vector.

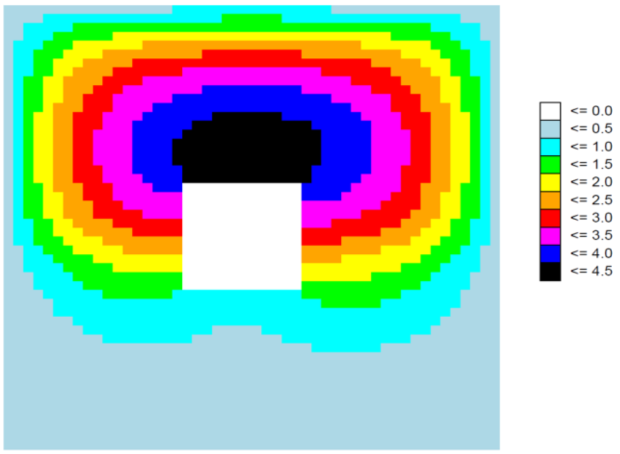
In this research we use a slightly more general form, inspired by the conduction of heat. First of all, the conductivity is non-homogeneous, in that a part of the area is isolated, so there is no heat conduction inside that area nor over its boundaries. Secondly, the forcing function f(x,y) is taken to be zero in one half of the rectangle and a constant positive value in the other half. This is expressed via the following equation:

$$\frac{\partial}{\partial x}K(x,y)\frac{\partial\Phi}{\partial x} + \frac{\partial}{\partial y}K(x,y)\frac{\partial\Phi}{\partial y} = f(x,y) \tag{2}$$

K(x,y) being the conductivity encapsulating the isolated area via K(x,y) = 0 inside and K(x,y) non-zero outside that area.

The reason to look into this extended problem is that it makes the numerical algorithm less regular: not all grid cells (see the next section) are identical.

The boundary conditions are chosen to be simple though: on the four sides of the rectangular area the temperature is taken to be zero. The heating term ensures that the temperature will be non-uniform (see Fig. 1).



<u>Figure 1</u>: Solution on a grid of 50 by 50 cells. The white rectangle in the middle is the isolated area, where the temperature remains at the starting value of 0.

In detail:

- The studied area is a rectangle with four open boundaries, $0 \le x \le W$ and $0 \le y \le H$.
- On the open boundaries the temperature is set to zero.

- The conductivity is uniform except inside a small rectangle defined by 2W/5 ≤ x ≤ 3W/5 and 2H/5 ≤ y ≤ 3H/5. Inside this area the conductivity is set to zero.
- The forcing term, the heating, is zero for 0 ≤ y ≤ H/2 and 1 for H/2 ≤ y ≤ H.

NUMERICAL ALGORITHM AND IMPLEMENTATION

The algorithm devised to solve this problem emphasizes the use of *arrays*, it is not meant as a practical or state-of-the-art method. By deliberately keeping the algorithm simple, we can easily study the access patterns and use various implementations.

First, the rectangular domain is split up into rectangular grid cells. Each grid cell is connected to its four neighbours by a discretised version of the equation. The solution is then approximated on this grid.

Second, the Poisson equation describes an equilibrium, This equilibrium can be numerically approximated by using finite differences and solving the resulting system of linear equations. However, by allowing the solution to develop over time, we get an algorithm that determines a new state in small steps and reaches the equilibrium solution in due time.

The algorithm looks like this:

- Set up the arrays to describe the mathematical equations (see below for details).
- Determine the change of the temperature per grid cell based on the current temperature in the central cell and the temperature of the four neighbours.
- When the change is known for all grid cells, apply it over the chosen time step and repeat until there is no significant change anymore.

By integrating Eq. 1 over a grid cell and approximating the gradients via finite differences, we obtain a discretised (algebraic) form. For a typical grid cell, the resulting equation to determine the change in temperature looks like this (see Fig. 2):

$$\begin{aligned}\Delta x \Delta y \delta T_c &= \Delta y K_w (T_w - T_c)/\Delta x + \Delta y K_e (T_e - T_c)/\Delta x + \\ &\quad \Delta x K_n (T_n - T_c)/\Delta y + \Delta x K_s (T_s - T_c)/\Delta y\end{aligned} \tag{3}$$

with the subscripts indicating which neighbour and the grid cells having a "volume" of Δx × Δy. In the equation the fluxes over the four sides are shown explicitly. For the edges of the isolated area the conductivity is actually zero.

By dividing by the "volume" of the grid cell, Δx × Δy, we get:

$$\begin{aligned}\delta T_c &= K_w (T_w - T_c)/\Delta x^2 + K_e (T_e - T_c)/\Delta x^2 + \\ &\quad K_n (T_n - T_c)/\Delta y^2 + K_s (T_s - T_c)/\Delta y^2\end{aligned} \tag{4}$$

So, the new temperature becomes:

$$T_c^{t+\Delta t} = T_c^t + \Delta t \cdot \delta T_c \tag{5}$$

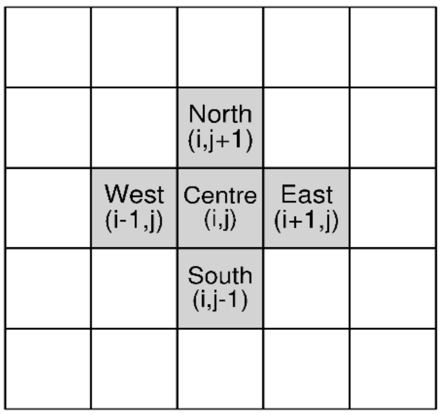


Figure 2: Schematic representation of the numerical method, the centre cell with its four neighbours.

The change per time step can be used to determine if the solution has converged already to the final temperature. As in this research we are not interested in the ultimate temperature distribution but rather the access patterns and the computational time, we do not discuss choosing the convergence criterium further. Instead we simply use a fixed number of iterations over the full grid.

Details of the Implementations

Four implementations (and several variants) were developed and they all use two-dimensional arrays, so that the four neighbours of the central cell with indices (i,j) have indices (i-1,j), (i+1,j) for the left and right neighours and (i,j-1), (i,j+1) for the lower and upper neighbours (see Fig. 2). These two-dimensional arrays are laid out in memory in essence as a long one-dimensional array, as illustrated in Fig. 3. This also shows the access pattern: all necessary data are present at strides of 1 or the size of the first (left-most) dimension.

One complication does arise and that is that we need extra rows and columns to cover the boundary conditions. These boundary cells do not change in value due to the heat conduction, but only because the boundary conditions prescribe such a change. In this research the boundary cells have a constant temperature of zero degrees.

A straightforward implementation is illustrated by this fragment of code (see SI for the full code):

```
du = 0.0 ! Reset the rate of change

do iy = 2,ny+1

do ix = 2,nx+1

if ( status(ix,iy) == 1 ) then

flux_west = 0.0

flux_east = 0.0

flux_north = 0.0

flux_south = 0.0

...
```

```
if ( status(ix-1,iy) /= 0 ) then
flux_west = dx * ( u(ix-1,iy) - u(ix,iy) )
endif
... other directions
endif
...
enddo
enddo
```

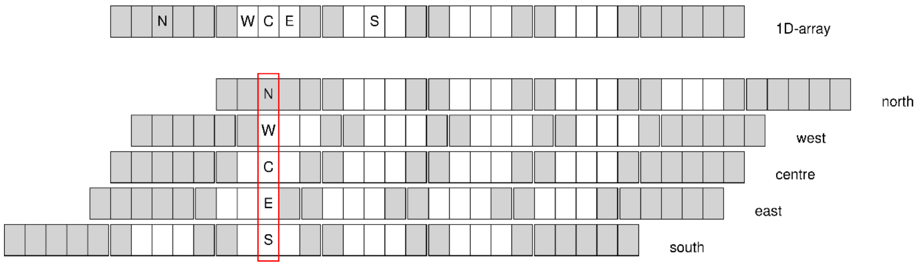


Figure 3: Sketch of the memory access for a 5 by 6 matrix (including the boundaries). The red bar indicates which array elements are involved for the calculation in cell (2,2). The array is shifted to align the respective array elements with this central cell. The light grey cells indicate the boundaries.

The indices `ix` and `iy`+ run from respectively 2 to nx+1 and 2 to ny+1, to avoid overflows (the shape of the matrices is (nx+2)×(ny+2)). The status of the cell is set to 1, if it is not part of the isolated area and to~0 if it is part of that area.

The constant `dx`+ encompasses the conductivity (`cond`)and the grid size in x-direction:

```
dx = cond / (x_extent/nx) ** 2
```

This "naïve" implementation turned out to be the least performant.

The second implementation encapsulates the condition that the cell and its neighbour are both conductive or not by introducing arrays `condx` and `condy` that are set to zero if either cell is non-conductive. Since two distinct directions are involved, we need two arrays. But this allows us to write the calculation per grid cell as:

```
du(ix,iy) = condx(ix-1,iy) * ( u(ix-1,iy) - u(ix,iy) ) &
+ condx(ix,iy) * ( u(ix+1,iy) - u(ix,iy) ) &
+ condy(ix,iy-1) * ( u(ix,iy-1) - u(ix,iy) ) &
+ condy(ix,iy) * ( u(ix,iy+1) - u(ix,iy) ) &
+ force(ix,iy)
```

with `du` the rate of change in the temperature (or whatever meaning we give the state variable).

The two other implementations use the Fortran features of *pointers* and *associate* blocks.

In the *pointer* solution we let pointers point to two-dimensional array sections:

```
pcentre => u(2:nx+1,2:ny+1)

pwest => u(1:nx ,2:ny+1)

...
```

and then the calculation can be done for the whole grid at once (written with separate flux contributions in mind):

```
do while ( ... )

!

! Calculate the change for the whole grid

!

pderiv = condw * (pwest - pcentre) + &

conde * (peast - pcentre) + &

condn * (pnorth - pcentre) + &

conds * (psouth - pcentre) + &

pforce

...

enddo
```

The solution with the *associate* feature is very similar, but the encompassing code looks like:

```
associate( &

pcentre => u(2:nx+1,2:ny+1), &

pwest => u(1:nx ,2:ny+1), &

peast => u(3:nx+2,2:ny+1), &

do while ( ... )

...

!

! Calculate the change for the whole grid

!

pderiv = condw * (pwest - pcentre) + &

conde * (peast - pcentre) + &

condn * (pnorth - pcentre) + &

conds * (psouth - pcentre) + &

pforce

...

enddo

end associate
```

This may require some explanation [6]:

- In Fortran *pointers* are not merely addresses to memory, but they carry more information about the variable or array they are pointing to, the array descriptor.

This makes it possible that a pointer selects a (non-contiguous) section from an array as shown here.

- The *associate* construct behaves as a smart macro facility. Rather than providing an alternative access to the memory, like pointers do, the construct serves to hide the ugly details of shifting the arrays so that the right elements are combined. This means, in principle, that the compiler does not need to worry about aliasing, as it has to in the case of pointers. Therefore, again in principle, more optimisations are possible.
- The code is more compact and details are hidden, if such array operations are used, but that also means that it is left entirely to the compiler how and in what order the array elements are accessed. This complicates a detailed analysis.

During the investigation several variants were developed on the basis of these four implementations, for instance, several versions that employ OpenMP loops to distribute the workload. Table 1 describes these in some detail.

Table 1: Variants of the basic program.

Indication	Description
Naïve	Use if-statements for the isolated area
Array	Instead of if-statements, arrays of conductivity used
Pointer	Use pointers to non-contiguous sections
Pointer with subroutine	Pass the pointers to a subroutine, so that the "pointer" attribute can be dropped
OpenMP	Most loops parallellised via OpenMP, less array operations
Strips	Matrix is split up into narrower strips for possibly better caching behaviour

Measuring and Comparing the Performance

All experiments involved running the programs over a wide range of matrix sizes, typically from 100×100 to 3000×3000, to examine any size dependency beyond merely the number of matrix elements. This led to the use of a scaled performance metric: the elapsed wall clock time or the CPU time is scaled with the size of the matrix -- the number of elements -- and then multiplied with a factor 10,000 to give values in the order of 1.

The elapsed time and the CPU time were measured via two Fortran intrinsic functions, system_clock and cpu_time.

Hardware and Software

The numerical experiments were done on a laptop with an Intel i7 processor, containing 8 P-cores and 8 E-cores, with a total of 24 threads. The performant (P) cores are to be preferred for compute-intensive programs. This can be arranged by setting the *affinity* of the program to the various cores, but that may be less than reliable, as sometimes much longer computation times resulted, than if the same computation was run again. This variation will be discussed in the "Results" section.

The operating system of the laptop was MicroSoft Windows 11.

Two different compilers were used to examine the performance properties of the various implementations and compiler options:

- The GNU Fortran compiler, version 14.1.0, indicated as *gfortran*.
- The Intel oneAPI Fortran compiler, version 2025.0.0, indicated as *ifx*.

The comparison in performance between the two compilers may be of some interest, but we considered the relative performance of programs built with the same compiler but different compiler options more important. Table 2 shows which compiler options were examined.

For the Intel oneAPI compiler, *ifx*, the option `-heap-arrays` was required, because of the way large temporary arrays are used -- either from the stack or from the heap. The Intel compiler also offers more control over the prefetching, but does so via a compiler directive.

Table 2: Overview of compiler options and compiler directives used in the experiments.

Category	*Gfortran*	*ifx*
Debug (no optimisation)	-g	-Od -heap-arrays
Level 2 optimisation	-O2	-O2 -heap-arrays
Level 3 optimisation	-O3	-O3 -heap-arrays
Host-specific instruction set	-O -march=native	-O2 -heap-arrays -Qxhost
OpenMP	-O2 -fopenmp	-O2 -Qopenmp
Prefetching	-O2 -fprefetch-loop-arrays	!dir$ prefetch *:1:8

PERFORMANCE RESULTS

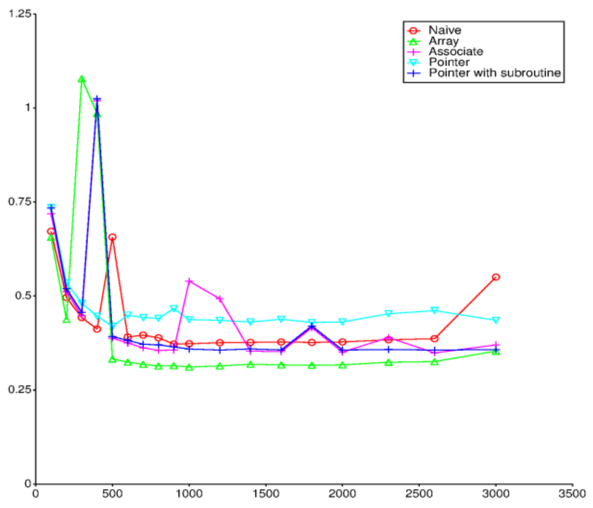
The experiments with the various implementations were mostly done using a square matrix. Typical results are seen in Figs. 4 and 5. For these graphs the timings were scaled by the number of matrix elements and a factor 10,000, to make the outcome in the order of 1.

As can be seen, the fastest implementation in both cases is the one using the array approach, where the isolation was incorporated in the conductivity, though for the *ifx* case the differences are small.

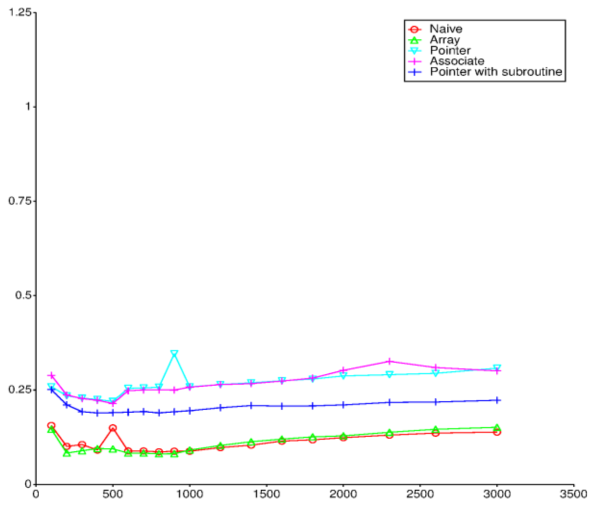
A possible explanation for the difference in performance for the pointer and associate approaches is that in these implementations non-contiguous array sections are used. This may lead to copies of these sections into temporary contiguous arrays. This explanation is supported by the fact that for the *ifx* compiler the compile option `-heap-arrays` turned out to be necessary.

Also note that the individual runs may be slower, leading to a higher time per grid cell, than would be expected from the surrounding data. This phenomenon occurred even when explicitly setting the *core affinity*. The only workaround for this seems to be to repeat the series, as sometimes several individual calculations on a row showed this anomalous timing (see SI).

When comparing the various implementations, the "pointer with subroutine" solution is generally faster than the "pointer" solution. So it pays to "strip off" the pointer attribute in this.



<u>Figure 4</u>: Results for the various implementations, using the *gfortran* compiler with the "host" option.



<u>Figure 5</u>: Results for the various implementations, using the *ifx* compiler with the "host" option

Dividing the Matrix in Strips

Most of the experiments were done with a square matrix, but to see what the separate effect of the *width* of the matrix might be, several experiments were done with a matrix with fixed width, but varying height and vice versa (see Figs. 6 and 7).

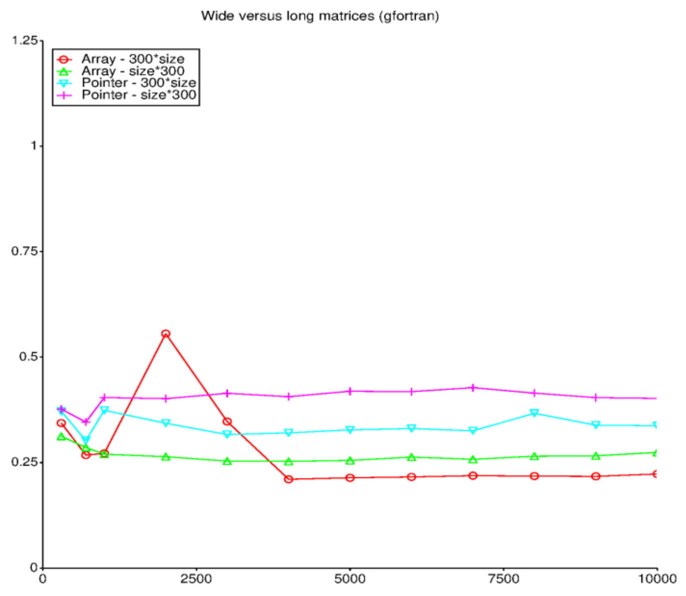


Figure 6: Timings for rectangular matrices, using the *gfortran* compiler.

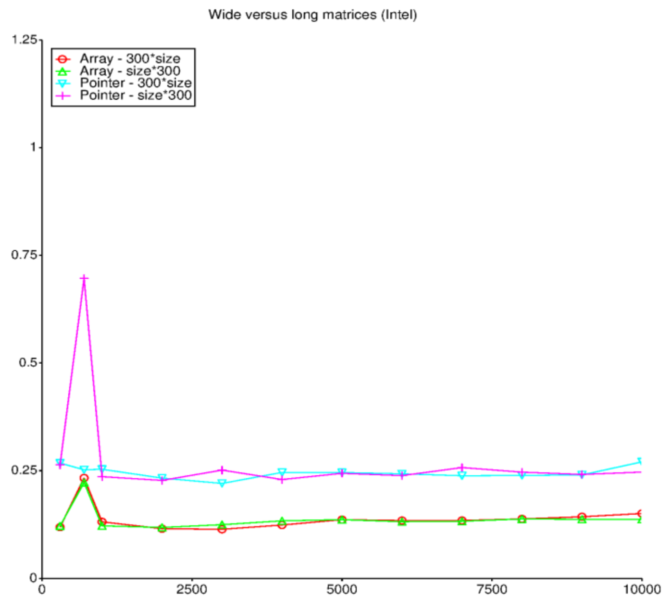


Figure 7: Timings for rectangular matrices, using the *ifx* compiler.

The *ifx* compiler shows fairly consistent results: the width and the height have no noticeable effect, the total size of the matrix is the determining factor. This is, however, not the case for the *gfortran* compiler. Here the width is of influence on the performance, the "narrow" matrices outperform the "wide" matrices by 15 to 25%.

This inspired the idea to develop yet another implementation: the matrix is split up into strips of roughly 200 elements (the choice was arbitrary) and then an extra loop is performed over the individual strips. This way, the arrays for the strips could be allocated independently and, in theory at least, the array accesses between strips would not need to interfere, that is, independent cache lines. The loop over the strips is easy to parallellise, though after each iteration, the boundaries need to be synchronised (see SI).

The use of these strips actually introduces an extra dimension to the problem -- a form of dimension lifting [2].

Parallellisation with OpenMP

As the algorithm we are dealing with is strongly based on *arrays*, it is also natural to look at the *OpenMP* facilities to accelerate the program. For a programmer there are various options to parallellise parts of the program [5,7]. When the main loop was parallellised and later the array operations were replaced by explicit loops (for easier parallellisation), the results were somewhat disappointing: with two or three threads the optimal performance is reached. Adding more threads did not lead to a faster calculation (see Figs. 8 and 9). In Fig. 10 the typical timings for the larger matrix sizes are compared against the number of threads:

- The two results for the ifx compiler show that a few threads affect the timings, but it levels out quickly. Moreover, dividing the matrix into strips is not beneficial: the performance is lower by roughly 20-30%.
- For the gfortran compiler the results are somewhat different, the "plain" implementation levels out at three threads, but the version with strips still gains some performance and is faster with four threads than the "plain" implementation.

According to the classical view of parallellising a program (Amdahl's law), we must split the program into a parallellised and a sequential part:

$$T_{total} = T_{sequential} + T_{parallel} / P$$

where P is the number of processors or threads.

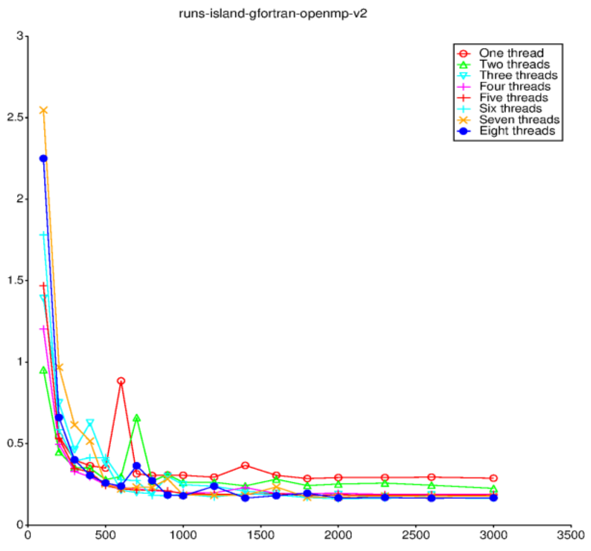


Figure 8: Results for the "strips" variant, using the *gfortran* compiler with the "openmp" option

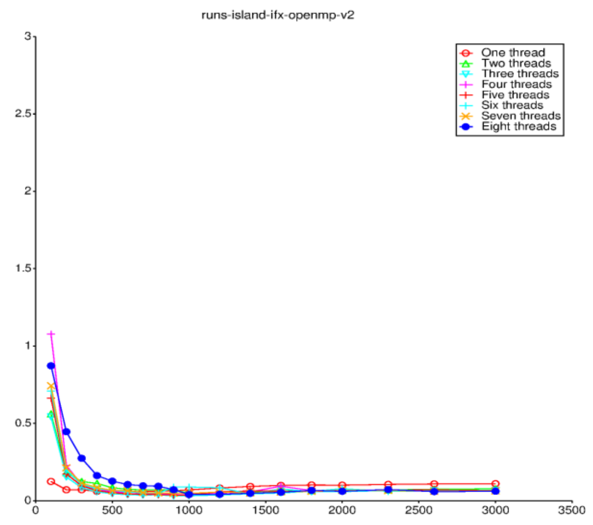


Figure 9: Results for the "strips" variant, using the *ifx* compiler with the "openmp" option.}

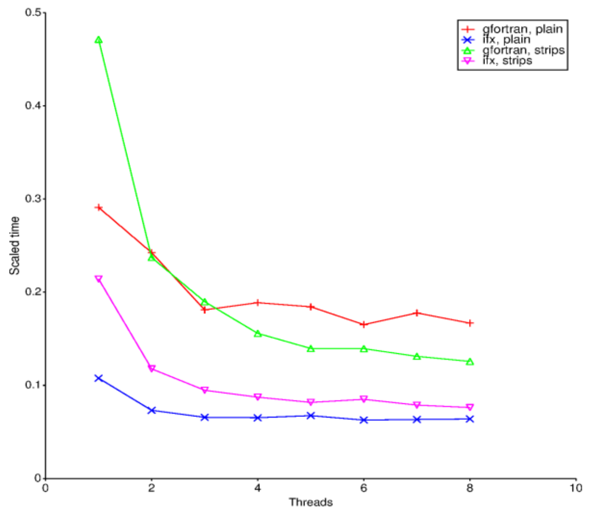


Figure 10: Timings as a function of the number of threads. The timings are the average over the largest three matrix sizes.

Since we compare the performance for the various matrix sizes by scaling with the number of elements in the matrix, the time consumed by the sequential part must consist of more than simply the overhead of starting and synchronising the threads, as that is expected to be independent of the matrix size. It must also consist of some manipulation of the arrays involved. Yet what that could be, cannot be determined from the source code alone, as all array manipulations for the "plain" implementation have been parallellised (see the SI) and the timings purely measure the overall iteration.

COMPARISON OF THE COMPILER OPTIONS

To get an impression of the effect of the compiler options, Table 3 shows the relative timing of the "array" implementation with respect to the timing for the level 2 optimisation (`-O2`). As is to be expected the debug option makes the program much slower, whereas the other options generally have only a marginal effect. Note, however, that some measurements (marked with an asterisk) showed spurious, non-reproducible values.

The relatively limited effect of the compile options may be due to the straightforward algorithm with a very regular access pattern using arrays in a double do-loop. Requiring prefetching may actually be obstructing the compiler, as the consequences for the program are unclear.

Table 3: Relative timing for different compiler options. Scaled to the results of "level 2 optimisation

Category	*Gfortran*	*ifx*
Debug (no optimisation)	3.3	14
Level 2 optimisation	**1**	**1**
Level 3 optimisation	0.5	1.0 (*)
Host-specific instruction set	1.0	1.2
OpenMP	1.0	0.6
Prefetching	1.0 (*)	2.0

CONCLUSIONS

In this study we have shown how the performance of an algorithm entirely based on two-dimensional arrays is affected by the implementation and by compiler options. While there is a significant difference between the *gfortran* compiler and the *ifx* compiler, in what implementation or compiler options would provide the best performance, we can draw at least the following general conclusions:

- Both compilers give the best results, that is, the fastest programs, when explicit loops over the arrays are used. This is not necessarily the most attractive option where the source code is concerned, as you have to specify the indices explicitly.
- Parallellisation via OpenMP has effect on the performance only for a small number of threads. With more than three or four threads, in most cases there is no gain. The exception is the implementation with strips and the *gfortran* compiler. Since the performance in this study is scaled by the number of matrix elements, there must be a fixed cost that scales with the matrix size, though this is not apparent from the source code.
- When measuring the performance, it should be taken into account that the performance of the program may be influenced arbitrarily: we have seen many runs where the performance was slower by a factor 5 or even 10 without there being a clear reason. This hinders this type of research (see the SI for an illustrative example).

Perhaps the most important conclusion is that if you want to make a reliable and fast library based on such algorithms, it pays to have various implementations tuned to the compiler at hand. Of course, this complicates the maintenance, so a second best approach is to use an implementation that shows consistent and predictable performance.

The algorithm presented here can also be transformed using the Mathematics of Arrays (MoA) approach [8]. The programs used here were all manually developed, based on the understanding of arrays in the Fortran language. But using MoA it should be possible to explore other implementations that give equivalent results but exploit the caching levels of the computer in a more efficient way, for instance via "dimension lifting". The fact that for the "strips" version of the program (Section 4.1) the results were ambiguous does not necessarily mean that with other size parameters, more fitting to the cache levels of the machine, we cannot do better.

Possible Solutions and Future Research

Ideally we propose a formal way to guarantee correctness to mathematics and optimal performance on arbitrary architectural platforms. This would be possible if a subset of Fortran functions were formulated in MoA. If that is done, array and tensor mathematics can be formulated in MoA, then formally reduced via Psi Reduction which composes all indices based on shapes, to a Denotational Normal Form (DNF). The DNF denotes the simplest representation of the algorithm with memory requirements to run on a uni-processor. The DNF is then formally transformed to an Operational Normal Form (ONF) in terms of starts stops and strides, or how to build this algorithm on a uni-processor with uniform memory. Now, because MoA views the machine as arrays also, the ONF can be dimension-lifted to match the machines components. For example, for an 8 by 6 array mapped to two processors, the new abstraction would be 2 by 4 by 6. Based on speed of the architecture chosen, cost functions can now be predicted. The only reason we would now need a compiler is to map the loops to components using general-purpose instructions common to all machines.

REFERENCES

[1] Gustafson, J.L.; Mullin, L.M.R. Tensors Come of Age: Why the AI Revolution will help HPC. CoRR **2017**, abs/1709.09108, [1709.09108].

[2] Leiserson, C.E.; Thompson, N.C.; Emer, J.S.; Kusmaul, B.C.; Lampson, B.W.; Sanchez, D.; Schardl, T.B. There’s Plenty of Room at the Top: What will drive computer performance after Moore’s Law? Science **2020**, 368. https://doi.org/10.1126/science.aam9744.

[3] Backus, J.W. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. Commun. ACM **1978**, 21, 613–641. https://doi.org/10.1145/359576.359579.

[4] Iverson, K.E. A Programming Language; John Wiley and Son: New York, 1962.

[5] Curcic, M. Modern Fortran, Building efficient parallel applications; Manning, 2020.

[6] Metcalf, M.; Reid, J.; Cohen, M.; Bader, R. Modern Fortran Explained, Incorporating Fortran 2023; Oxford University Press, 2023.

[7] Hermanns, M. Parallel Programming in Fortran 95 using OpenMP, 2002.

[8] Mullin, L.; Raynolds, J. Conformal Computing: Algebraically connecting the hardware/software boundary using a uniform approach to high-performance computation for software and hardware applications **2008**.