What Happens to a Dream Deferred?
Chasing Automatic Offloading in Fortran 2023

Damian Rouson
Computer Languages and Systems Software Group

International Workshop on Automatic Performance Tuning (iWAPT), 31 May 2024
Overview

From Software Archaeology to Software Modernity

01 Background
02 Motivation
03 Parallelism in Fortran 2023
04 AI

05 HPC
06 Ruminations
“Harlem”
By Langston Hughes, 1951

What happens to a dream deferred?
Does it dry up
like a raisin in the sun?
Or fester like a sore—
And then run?
Does it stink like rotten meat?
Or crust and sugar over—
like a syrupy sweet?

Maybe it just sags
like a heavy load.

Or does it explode?
The Fortran Automatic Coding System for the IBM 704, the first programmer’s reference manual for Fortran

(Public Domain)
“Fortran is a new and exciting language used by programmers to communicate with computers. It is exciting as it is the wave of the future.”

Character of Dorothy Vaughan, a NASA mathematician and programmer, as played by Octavia Spencer in *Hidden Figures* (20th Century Fox, 2016).
Overview

From Software Archaeology to Software Modernity

01  Background
02  Motivation
03  Parallelism in Fortran 2023
04  AI

05  HPC
06  Ruminations
1977 Turing Award Lecture:
“Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”

Rumors of Fortran’s Demise…

2 Programming Alternatives

In 1984, McGraw noted that by all indications future supercomputers would be multiprocessors. Today, most supercomputer users and vendors agree. But can programmers take advantage of the horse-

model to the imperative model of Fortran. To begin, we list the desired characteristics of a true parallel programming language [I]:

1. The language must insulate the programmer from the underlying machine. Deriving and expressing a parallel algorithm is hard enough; one should not have to reprogram it for each new machine.

2. Parallelism must be implicit in the semantics of the language. The compilation system should not have to unravel the behavior of the compu-

tation.

3. When a programmer desires determinacy, the language should guarantee it. Regardless of the conditions of execution, a program that realizes a determinate algorithm should yield the same results for the same data.

Of the three items, the last is an issue only when automatic parallelizing compilers are not available and the programmer is responsible for expressing and managing parallelism. Programmers will make mistakes, and those mistakes may remain hidden until system activity changes the rate of execution. This is all we will say about determinacy, as most parallel ma-

chines support automatic parallelizing compilers.

Regarding the first two items, however, imperative languages fail to meet the requirements. Remember that languages like Fortran were designed to exploit von Neumann machines. As such their computational model assumes that a single program counter will step

For example, consider the following Fortran ex-

ccerpt:

\begin{align*}
A & = \text{Foo}(X) \\
B & = \text{Goo}(Y)
\end{align*}

Determining if these statements can execute in parallel requires a full understanding of both functions. Because of COMMON blocks, they might share data.

Further, because of aliasing, some combination of X, Y, A, or B might represent the same memory cell. Hence the parallelism in this excerpt is not immediately obvi-

ous, and its discovery requires interprocedural analysis or function expansion.

Functional languages, on the other hand, meet all the requirements listed above and do not require anal-

ysis for the discovery of parallelism [I,11,13,14]. A functional program is a collection of mathematically sound expressions comprised of both intrinsic and user defined functions. These functions are well defined and deterministic. That is, they define a unique mapping between their domain and their range. A function passed the same set of values will yield the same results regardless of the environment of invocation. This es-

tablishes referential transparency, which implies that the evaluation of an expression, or the sharing of its subexpressions, does not change the value it denotes. Consequently, expressions are side effect free. The concept of a Fortran COMMON block does not ex-

ist. In the absence of side effects, programmers cannot see the target machine; the concept of data replaces memory, and the concept of creation replaces update.

Further, in the absence of side effects, programs are implicitly parallel.
Or a Roadmap for Fortran’s Future?
Overview

From Software Archaeology to Software Modernity

01 Background
02 Motivation
03 Parallelism in Fortran 2023
04 AI

05 HPC
06 Ruminations
Explicit Parallelsim in Fortran 2023

Single Program Multiple Data (SPMD) parallel execution
— Synchronized launch of multiple “images” (process/threads/ranks)
— Asynchronous execution except where program explicitly synchronizes
— Error termination or synchronized normal termination

```fortran
1 program main
2   implicit none
3   print *,"Hello from image ", this_image(), "of", num_images()
4 end program
```
SPMD Execution Sequence

1. After the creation of a fixed number of images, each image’s first “segment” (sequence of statements) executes.
2. Image control statements totally order segments executed by a single image and partially order segments executed by separate images.
Partitioned Global Address Space (PGAS)

Coarrays:
- Distributed data structures — greeting
- Facilitate Remote Memory Access (RMA) — line 15

```fortran
program main
  !! One-sided communication of distributed greetings
  implicit none
  integer, parameter :: max_greeting_length=64, writer = 1
  integer image
  character(len=max_greeting_length) :: greeting, image
  associate(me => this_image(), ni=>num_images())
  write(greeting,*), "Hello from image", me,"of",ni ! local (no "[]")
  sync all ! image control
  if (me == writer) then
    do image = 1, ni
      print *,greeting[image] ! one-sided communication: "get"
    end do
    end if
  end associate
end program
cd fortran
make run-hello
```
Teams of images can be formed at runtime.

Collective subroutines: \( \text{co}_{\{\text{broadcast, sum, max, min, reduce}\}} \)

Atomic subroutines:
- \( \text{atomic}_{\{\text{define, ref, add, fetch_add, …}\}} \)
- Events: counting semaphores with post/wait/query operations

Failed/stopped image detection, locks, critical sections, …
Coarray Fortran began as a syntactically small extension to Fortran 95:
— Square-bracketed “cosubscripts” distribute & communicate data
Integration with other features:
— Array programming: colon subscripts
— OOP: distributed objects
Minimally invasive:
— Drop brackets when not communicating
Communication is explicit:
— Use brackets when communicating
PRIF

- Enable a compiler to target multiple implementations of PRIF
  - I.e. enable a vendor to supply their own parallel runtime
- Enable a PRIF implementation to be used by multiple compilers
- Isolate a compiler’s support of the parallel features of the language from any particular details of the communication infrastructure
- Our group’s experience with UPC and OpenCoarrays has shown this to be valuable
Caffeine leverages GASNet-EX, a high-performance networking middleware that undergirds a broad ecosystem of languages, libraries, frameworks, and applications.
## Overview

From Software Archaeology to Software Modernity

<table>
<thead>
<tr>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>Motivation</td>
<td>Parallelism in Fortran 2023</td>
<td>AI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>05</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC</td>
<td>Ruminations</td>
</tr>
</tbody>
</table>
Implicit Parallelism

In addition to the SPMD/PGAS features that work in shared or distributed memory, several features facilitate expressing unordered sets of calculations amenable to multithreading, vectorization, or accelerator offloading:

```
  do concurrent + pure procedures, including elemental procedures

  integer, row, col :: window=4, time=1
  integer, parameter :: window=4, time=1

  associate(rows => size(distance%body, 1), cols => size(distance%body, 2))
  do concurrent(row=1:rows, col=1:cols)
    associate(first_row => max(1, row-window), last_row => min(row+window, rows))
    distance%body(row, col) = minval(hypot(&
      this%body(first_row=last_row, time) - rhs%body(row, time), &
      this%body(first_row=last_row, col) - rhs%body(row, col) &
    ))
  end associate
  end do
end associate
```

where statement

Array statements + elemental procedures (intrinsic or user-defined):
matmul, reduce, transpose, dot_product, merge, pack, unpack, count, any, all, findloc, ...
Inference-Engine

Use case:

Goals:
- To explore language-based parallelism, including GPU offloading.
- To simplify the workflow for training neural networks, i.e., eliminate the telephone game.

How:
- A functional programming style that facilitates concurrent inference across a large collection of inputs using multiple specialized neural networks.
- A training algorithm that squeezes out most unnecessary programmer-imposed ordering of input data.

Discussion
Run ICAR and save training data. Import training data into PyTorch and train neural network. Run nexport to export network to JSON. Import network into ICAR via Inference-Engine and validate.
Inference-Engine

Use case:

Goals:
- To explore language-based parallelism, including GPU offloading.
- To simplify the workflow for training neural networks, i.e., eliminate the telephone game.

How:
- A functional programming style that facilitates concurrent inference across a large collection of inputs using multiple specialized neural networks.
- A training algorithm that squeezes out most unnecessary programmer-imposed ordering of

Runtime Training in ICAR with embedded Inference-Engine

Rinse, Repeat…
FASTGPT: FASTER THAN PYTORCH IN 300 LINES OF FORTRAN

March 14, 2023
Authors: Ondřej Čertík, Brian Beckman

In this blog post I am announcing fastGPT, fast GPT-2 inference written in Fortran. In it, I show

1. Fortran has speed at least as good as default PyTorch on Apple M1 Max.
2. Fortran code has statically typed arrays, making maintenance of the code easier than with Python
3. It seems that the bottleneck algorithm in GPT-2 inference is matrix-matrix multiplication. For physicists like us, matrix-matrix multiplication is very familiar, unlike other aspects of AI and ML. Finding this familiar ground inspired us to approach GPT-2 like any other numerical computing problem.
4. Fixed an unintentional single-to-double conversion that slowed down the original Python.
5. I am asking others to take over and parallelize fastGPT on CPU and offload to GPU and see how fast you can make it.

About one month ago, I read the blogpost GPT in 60 Lines of NumPy, and it piqued my curiosity. I looked at the corresponding code (picoGPT) and was absolutely amazed, for two reasons. First, I hadn't known it could be so simple to implement the GPT-2 inference. Second, this looks just like a typical computational physics code, similar to many that I have developed and maintained throughout my career.

https://tinyurl.com/fastgpt-by-certik
do k=1,lev
  do j=1,lon
    do i=1, lat
      outputs(i,j,k) = inference_engine%infer(inputs(i,j,k))
    end do
  end do
end do
end do

do concurrent(i=1:lat, j=1:lon, k=1:lev)
  outputs(i,j,k) = inference_engine%infer(inputs(i,j,k))
end do

outputs = inference_engine%infer(inputs) ! elemental
Motility Analysis of T-Cell Histories in Activation (Matcha)

A parallel virtual T-cell model.

Matcha tracks the stochastic T-cell motions according to multiple distributions of speeds and angles, accounting for the dependence of speed on the turning angle and on the previous speed.

T cells must mount a coordinated attack in order to avoid overwhelming the host tissue.

The study of T-cell/T-cell interactions remains in its infancy [1].

Some communication occurs via secreting soluble mediators, e.g., cytokines and chemokines.

Matcha models mediator spread via a 3D diffusion equation:

\[ \phi_t = D \nabla^2 \phi \]

where \( \phi_t = \partial \phi / \partial t \).

Heat Equation

\[
\frac{\partial T}{\partial t} = \alpha \nabla^2 T
\]

\[
\{T\}^{n+1} = \{T\}^n + \Delta t \cdot \alpha \cdot \nabla^2 \{T\}^n
\]

\[ T = T + dt \cdot \alpha \cdot \text{laplacian}.T \]

local objects

pure user-defined operators
Explicitly pure procedures

- Side-effect free: no I/O, no `stop`, no image control, etc.
- Functions: `intent(in)` arguments
- Subroutines: specified argument `intent`
- Deterministic in most cases (Fortran 2002X simple removes most non-determinism)

Implicitly pure procedures: `elemental`

Associate

- Define immutable state by associating with an expression, e.g., function reference.
- Only pure procedures may be invoked inside a `do concurrent` block.
- Every intrinsic function is pure

Error termination in pure procedures

Variable `stop` codes

Use objects to encapsulate multiple entities in one function results.

A Functional Programming Pattern

```fortran
function functional_matches_procedural() result(test_passes)
  logical test_passes
  integer, parameter :: steps = 6000, n=32
  real, parameter :: tolerance = 1.E-06, alpha = 1.
  real, parameter :: side=1., boundary_val=1., internal_val=2.
  associate( T_f => T_functional(), T_p => T_procedural() )
  associate( L_infinity_norm => maxval(abs(T_f - T_p)) )
  test_passes = L_infinity_norm < tolerance
  end associate
  end associate

contains

function T_functional()
  real, allocatable :: T_functional(:,:,:)
  type(subdomain_t), save :: T[*]
  integer step

  call T%define(side, boundary_val, internal_val, n)

  associate(dt => T%dx()*T%dy()/(4*alpha))
  do step = 1, steps
    sync all
    T = T + dt * alpha * .laplacian. T
  end do
  end associate

  T_functional = T%values()
end function
```
Halo Exchange

```
116  real(rkind), allocatable :: halo_x(:,:,):[:]
117  integer, parameter :: west=1, east=2

134  me = this_image()
135  num_subdomains = num_images()
137  my_nx = nx/num_subdomains + merge(1, 0, me <= mod(nx, num_subdomains))

232  subroutine exchange_halo(self)
233    class(subdomain_2D_t), intent(in) :: self
234    if (me>1) halo_x(east,:)[me-1] = self%s_(1,:)
235    if (me<num_subdomains) halo_x(west,:)[me+1] = self%s_(my_nx,:)
236  end subroutine
```
Loop-Level Parallelism

188 do concurrent(j=2:ny-1)
189     laplacian_rhs%(i, j) = &
190     (halo_left(j) - 2*rhs%(i, j) + rhs%(i+1,j))/dx**2 + &
191     (rhs%(i, j-1) - 2*rhs%(i, j) + rhs%(i,j+1))/dy**2
192 end do
Purely functional parallel algorithms (user-defined operators) operating on distributed objects (derived type coarrays) with automatic GPU offloading via do concurrent.
Compiler Status

Supporting CAF features:

- Cray
- Intel
- GNU
- NAG

Automatic offloading of do concurrent:

- NVIDIA
- Intel
- Cray

LLVM Flang:

- Parses and verifies CAF syntax and semantics
- Does not yet lower CAF features

Berkeley Lab develops

- Frontend unit tests for CAF features
- Frontend bug fixes
- Caffeine: a candidate parallel runtime
- PRIF: a specification
The World’s Shortest Bug Reproducer

end
Overview

From Software Archaeology to Software Modernity

01 02 03 04
Background  Motivation  Parallelism in Fortran 2023  AI

05 06
HPC  Ruminations
<table>
<thead>
<tr>
<th>Rank</th>
<th>Change</th>
<th>Language</th>
<th>Percentage</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Python</td>
<td>16.33%</td>
<td>+2.88%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>C</td>
<td>9.98%</td>
<td>-3.37%</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C++</td>
<td>9.53%</td>
<td>-2.43%</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Java</td>
<td>8.69%</td>
<td>-3.53%</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>C#</td>
<td>6.49%</td>
<td>-0.94%</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>JavaScript</td>
<td>3.01%</td>
<td>+0.57%</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Visual Basic</td>
<td>2.01%</td>
<td>-1.83%</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Go</td>
<td>1.60%</td>
<td>+0.61%</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>SQL</td>
<td>1.44%</td>
<td>-0.03%</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Fortran</td>
<td>1.24%</td>
<td>+0.46%</td>
</tr>
</tbody>
</table>
The main reason for Fortran’s resurrection is the growing importance of numerical/mathematical computing. Despite lots of competitors in this field, Fortran has its reason for existence. Let’s briefly check the competition out. Python: choice number one, but slow, MATLAB: very easy to use for mathematical computation but it comes with expensive licenses, C/C++: mainstream and fast, but they have no native mathematical computation support, R: very similar to Python, but less popular and slow, Julia: the rising new kid on the block, but not mature yet. And in this jungle of languages, Fortran appears to be fast, having native mathematical computation support, mature, and free of charge. Silently, slowly but surely, Fortran is making its way back into the foreground. It is surprising but undeniable. --Paul Jansen CEO TIOBE Software
Sometimes it sags like a heavy burden.

Sometimes it explodes in a segmentation fault!

Sometimes it explodes in popularity.

Let’s hope the popularity maintains and realizes the dream.
Acknowledgements

The Berkeley Lab Fortran Team
Dan Bonachea, Hugh Kadhem, Brad Richardson, Kate Rasmussen

Past and Present Collaborators
Jeremy Bailey, David Torres, Kareem Jabbar Weaver, Jordan Welsman, Yunhao Zhang
The Problem is Not Fortran

Damian Rouson

Computer Languages and Systems Software (CLaSS) Group

NUCLEI Meeting, 29 May 2024
Popularity and Use
- Tiobe Index
- NERSC Data
- Open-Source: fpm, Caffeine, Veggies, Rojff
- Growth in Compilers: LFortran, LLVM Flang, ...

Fortran 2023 by Example
- Fusion
- Weather
- Climate
- FFTs, Multigrid, etc.

So what are the Problems?
- Perception
- Geography/Culture
- State of Practice
- State of Compilers
Compiled languages used at NERSC

- Fortran remains a common language for scientific computation.
- Noteworthy increases in C++ and multi-language
- Language use inferred from runtime libraries recorded by ALTD. (previous analysis used survey data)
  - ALTD-based results are mostly in line with survey data.
  - No change in language ranking
  - Survey underrepresented Fortran use.
- Nearly ¼ of jobs use Python.

Source: B. Austin et al., NERSC-10 Workload Analysis, 2020, doi:10.25344/S4N30W.
CAF at Scale: Magnetic Fusion

Application focus:
— The shift phase of charged particles in a tokamak simulation code

Programming models studied:
— CAF + OpenMP or
— Two-sided MPI + OpenMP

Highlights:
— Experiments on up to 130,560 processors
— 58% speed-up of the CAF implementation over the best multithreaded MPI shifter algorithm on largest scale
— “the complexity required to implement ... MPI-2 one-sided, in addition to several other semantic limitations, is prohibitive.”

CAF at Scale: Weather

Application:
— European Centre for Medium Range Weather Forecasts (ECMWF) operational weather forecast model

Programming models studied:
— CAF or
— Two-sided MPI

Highlights:
— Simulations on > 60K cores
— Performance improvement from switching to CAF peaks at 21% around 40K cores
Application:
— Intermediate Complexity Atmospheric Research (ICAR) model
— Regional impacts of global climate change

Programming models studied:
— CAF over one-sided MPI
— CAF over OpenSHMEM
— Two-sided MPI
— Cray CAF

Highlights:
— “... we used up to 25,600 processes and found that at every data point OpenSHMEM was outperforming MPI.”
— “The coarray Fortran with MPI backend stopped being usable as we went over 2,000 processes... the initialization time started to increase exponentially.”
CAF at Scale: CFD, FFTs, Multigrid

Applications studied:

- Magnetohydrodynamics (MHD)
- 3D Fast Fourier Transforms (FFTs) used in infinite-order accurate spectral methods
- Multigrid methods with point-wise smoothers requiring fine-grained messaging

Programming models studied:

- CAF or
- One-sided MPI-3

Highlights:

- Simulations on up to 65,536 cores
- “CAF either draws level with MPI-3 or shows a slight advantage over MPI-3.”
- “CAF and MPI-3 are shown to provide substantial advantages over MPI-2.
- “CAF code is of course much easier to write and maintain…”