SwarmStorm: A Multiprocessor Satellite Simulation Demonstration Using Fortran Coarrays

Daniel Topa

December 6, 2024

Abstract

SwarmStorm is a Fortran-based multiprocessor simulation for satellite interactions, leveraging coarray technology to enable efficient parallel computation. This report outlines the critical features of the program, including its object-oriented design with the 'satellite' type and its operations, and demonstrates how coarrays streamline distributed memory computations in the context of satellite simulations.

Contents

1 Introduction **2**

2 Comparing Fortran and C++ for Linear Algebra Operations **2**
- 2.1 Dot Product . . . 2
- 2.2 Matrix-Vector Multiplication . . . 2

3 Key Features of the Satellite Module **3**
- 3.1 Satellite Type . . . 3
- 3.2 Type-Bound Procedures . . . 3

4 Coarray Implementation **3**

5 Side-by-Side Comparison: MPI vs Coarray **3**
- 5.1 Image Teams for Coarrays . . . 4
  - 5.1.1 What is an Image Team? . . . 5
  - 5.1.2 Application to Satellite Simulations . . . 5
- 5.2 Image Teams for Coarrays . . . 5
  - 5.2.1 Concept of Image Teams . . . 5
  - 5.2.2 Forming and Using Teams . . . 5
  - 5.2.3 Application to Satellite Simulations . . . 6
  - 5.2.4 Advantages of Image Teams . . . 6
  - 5.2.5 Conclusion . . . 6
- 5.3 Image Teams in Coarrays . . . 6
  - 5.3.1 Concept of Image Teams . . . 6
  - 5.3.2 Creating Image Teams . . . 7

5.3.3 Using Image Teams 7
5.3.4 Synchronization Between Teams 7
5.3.5 Applications of Image Teams 7
5.3.6 Best Practices 8

6 Conclusion **8**

7 Fortran Readings **8**

A Proof of Principle **9**

1 Introduction

SwarmStorm employs intrinsic Fortran tools to simulate the dynamic interactions of satellite constellations. The program leverages coarrays to distribute workload across multiple processors efficiently, enabling detailed simulations of satellite parameters, resource management, and orbital dynamics. This report highlights key features of the program, focusing on its object-oriented design and the application of coarrays.

2 Comparing Fortran and C++ for Linear Algebra Operations

2.1 Dot Product

In Fortran:

```
real(dp) :: x(3), y(3), result
result = dot_product(x, y)
```

Fortran handles vector operations efficiently with intrinsic functions like `dot_product`, making the implementation compact and intuitive.

In C++:

```
double dot_product = 0.0;
for (size_t i = 0; i < x.size(); ++i) {
    dot_product += x[i] * y[i];
}
```

In C++, the same operation requires explicit iteration through the elements, which provides control but is more verbose.

2.2 Matrix-Vector Multiplication

In Fortran:

```
real(dp) :: A(3, 3), x(3), b(3), r(3)
r = matmul(A, x) - b
```

Fortran's array syntax and `matmul` intrinsic simplify matrix-vector operations.

In C++:

```cpp
std::vector<double> r(x.size(), 0.0);
for (size_t i = 0; i < A.size(); ++i) {
    for (size_t j = 0; j < x.size(); ++j) {
        r[i] += A[i][j] * x[j];
    }
    r[i] -= b[i];
}
```

C++ requires explicit loops for matrix-vector multiplication, which is verbose but flexible for more complex customizations.

3 Key Features of the Satellite Module

The ‘satellite‘ module, ‘mClassSatellite‘, defines the core satellite type and its associated operations. Below are the critical features of the module:

3.1 Satellite Type

The ‘satellite‘ type encapsulates key attributes such as mass, fuel, orbital radius, and UUID. It also includes type-bound procedures for parameter setting and listing.

```fortran
    type :: satellite
        ! rank 2
        integer ( kind = ip ), dimension ( 1 : UUID_SIZE ) :: uuid
        character ( kind = kindA, len = 36 ) :: moniker, model
        ! rank 1
        real ( kind = rp ) :: mass_metal   = 0.0_rp, mass_fuel      = 0.0_rp, fuel_burn_rate = 0.0_rp
        real ( kind = rp ) :: radius_orbit = 0.0_rp, energy_kinetic = 0.0_rp
        real ( kind = rp ) :: time_born    = 0.0_rp, time_end       = 0.0_rp, time_attack    = 0.0_rp
        integer ( kind = ip ) :: index, color

    contains

        ! Type-bound procedures
        ! PROCEDURE inside a type defines how the type behaves and ties methods directly to it.
        procedure, public :: list_satellite_parameters    => list_satellite_parameters_sub, &
                             set_satellite_parameters     => set_satellite_parameters_sub
    end type satellite
```

3.2 Type-Bound Procedures

The type-bound procedures, such as `set_satellite_parameters` and `list_satellite_parameters`, provide robust interfaces for managing satellite objects.

SwarmStorm employs intrinsic Fortran tools to simulate the dynamic interactions of satellite constellations. The program leverages coarrays to distribute workload across multiple processors efficiently, enabling detailed simulations of satellite parameters, resource management, and orbital dynamics. This report highlights key features of the program, focusing on its object-oriented design and the application of coarrays.

4 Coarray Implementation

Coarrays in Fortran enable parallelization by extending the language with syntax for distributed arrays. SwarmStorm uses coarrays to represent satellite teams, distributing computational workloads across multiple processors. The following example demonstrates the coarray implementation:

```
real, allocatable :: position[:]
allocate(position[*])
position[this_image()] = compute_position()
sync all
```

This approach ensures efficient synchronization and communication between processors, facilitating high-performance simulations.

5 Side-by-Side Comparison: MPI vs Coarray

MPI Code	Coarray Code

`program mpi_hello_world`
 	
`use mpi`
`implicit none`
 	
`call MPI_Init(ierr)`
`call`
`MPI_Comm_rank(MPI_COMM_WORLD,`
`rank, ierr)`
`call`
`MPI_Comm_size(MPI_COMM_WORLD,`
`size, ierr)`
 	
`print *, 'Hello, World from`
`process ', &`
`rank, ' out of ', size`
 	
`call MPI_Finalize(ierr)`
`end program mpi_hello_world`
	`program coarray_hello_world`
	
	`! No module needed`
	`implicit none`
	
	`num_images = num_images()`
	`my_image = this_image()`
	
	`print *, 'Hello, World from`
	`image ', &`
	`my_image, ' out of ',`
	`num_images`
	
	`end program coarray_hello_world`

Table 1: Comparison of MPI and Coarray implementations for a Hello World program.

Compilation and Execution: To compile and execute the MPI and coarray programs:

```
# For MPI:
mpifort -o mpi_hello_world mpi_hello_world.f90
mpirun -np 4 ./mpi_hello_world
```

```
# For Coarrays:
caf -o coarray_hello_world coarray_hello_world.f90
cafrun -n 4 ./coarray_hello_world
```

5.1 Image Teams for Coarrays

Image Teams for Coarrays

In Fortran coarray programming, image teams provide a mechanism for grouping images into smaller, independent subgroups, allowing for localized operations and reduced communication overhead. This concept is particularly useful in simulations involving multiple entities, such as satellite packs, where different groups can operate independently or engage in coordinated interactions. Definition and Benefits

5.1.1 What is an Image Team?

An image team is a subset of the program's coarray images that can collaborate independently of other teams. Key advantages include:

Scalability: Teams enable efficient use of computational resources by limiting communication to within the team. Flexibility: Teams can be dynamically formed and dissolved as simulation needs evolve. Isolation: Different teams operate in isolation, preventing interference and ensuring localized computation.

5.1.2 Application to Satellite Simulations

In satellite engagement scenarios, image teams can represent:

Allied Satellite Packs: Each pack forms a team, coordinating among themselves for maneuvers or data sharing. Enemy Satellite Packs: Similarly, enemy satellites can form their own teams. Engagement Regions: Teams can represent satellites within a particular spatial or operational domain.

For instance:

A team of 5 friendly satellites can be defined for coordinated defense. Another team of 3 enemy satellites can operate independently for reconnaissance.

5.2 Image Teams for Coarrays

In Fortran coarray programming, **image teams** provide a mechanism for dividing the images into smaller, independent groups. This capability is essential for managing the complexity of parallel tasks, especially in simulations that involve independent but concurrent operations.

5.2.1 Concept of Image Teams

Each image in Fortran can belong to one or more teams, where a team is a subset of the total number of images. By organizing images into teams, a developer can effectively partition the computational resources, enabling finer control over task parallelism.

For example, in a simulation of satellite engagements:

- Each team of images can represent a *pack of satellites*.
- Teams operate independently, such that operations on one team do not interfere with another.
- This independence allows for scalable and efficient simulations.

5.2.2 Forming and Using Teams

The formation of image teams in Fortran uses the `FORM TEAM` and `CHANGE TEAM` constructs. Here is an example:

```
FORM TEAM (team_number, my_new_team)
CHANGE TEAM (my_new_team)
! Code that operates within the new team
END TEAM
```

5.2.3 Application to Satellite Simulations

In the context of satellite simulations:

1. **Team Assignment:** Each team corresponds to a specific mission or group of satellites, such as a reconnaissance team or an orbital defense team.
2. **Independent Execution:** Teams can operate on different timelines or perform entirely different sets of computations without cross-team dependencies.
3. **Scalability:** Teams allow simulations to scale efficiently across available processors.

5.2.4 Advantages of Image Teams

- **Reduced Synchronization Overhead:** Teams minimize unnecessary synchronization across unrelated tasks, improving performance.
- **Encapsulation of Tasks:** By isolating tasks to specific teams, developers can simplify the logic and improve maintainability.
- **Parallel Debugging:** Errors can be confined to a team, making parallel debugging more manageable.

5.2.5 Conclusion

The concept of image teams in Fortran coarrays empowers developers to tackle complex parallel programming challenges with structured and efficient solutions. In satellite simulations, they allow for a clear and scalable organization of computational tasks, enabling realistic and high-performance models of satellite behavior.

5.3 Image Teams in Coarrays

In Fortran coarray programming, *image teams* provide a powerful mechanism to hierarchically organize images for finer control over parallelism and communication. Image teams are particularly useful when subsets of images need to work independently or collaboratively on specific tasks, such as domain decomposition or independent simulations.

5.3.1 Concept of Image Teams

An *image team* is a subset of the default set of images. Within a team:

- Images are indexed locally, starting from 1.
- Communication and synchronization operations are restricted to the images within the team.
- Teams operate independently of one another, enabling parallel tasks without interference.

Image teams enable use cases such as:

- **Domain decomposition:** Dividing a computational domain into spatial regions, with each team handling a specific region.
- **Independent tasks:** Assigning different algorithms or computations to separate teams.
- **Hierarchical parallelism:** Combining multiple levels of parallel computation for greater scalability.

5.3.2 Creating Image Teams

Image teams are created using the `FORM TEAM` statement. For example:

```
integer :: team_number
type(team_type) :: my_team

! Assign team membership based on image index
team_number = mod(this_image(), 2) + 1

! Form image teams
form team (team_number, my_team)
```

Listing 1: Forming image teams

In this example:

- Images are divided into two teams based on whether their image indices are even or odd.
- Each team has its own independent scope for computation and synchronization.

5.3.3 Using Image Teams

Once a team is formed, all subsequent coarray operations within the `CHANGE TEAM` construct are scoped to that team:

```
change team (my_team)
    ! Operations here involve only 'my_team' members
    print *, 'Image', this_image(), 'of team', num_images()
end team
```

Listing 2: Using image teams

Inside a `CHANGE TEAM` block:

- `THIS_IMAGE()` and `NUM_IMAGES()` return indices and size local to the team.
- Synchronization operations such as `SYNC ALL` apply only to images within the team.

5.3.4 Synchronization Between Teams

While teams operate independently, inter-team communication requires switching back to the parent team using `SYNC TEAM` and coarray assignments or other synchronization mechanisms.

5.3.5 Applications of Image Teams

Image teams are well-suited for:

- **Parallel Discrete Event Simulation (PDES):** Assigning each team a different "causal bubble" or event set.
- **Satellite Engagement Models:** Dividing satellite packs into separate teams to simulate domain-specific interactions.
- **Multi-resolution Algorithms:** Running different levels of resolution on separate teams and combining results in the parent team.
- **Load Balancing:** Allocating more images to computationally expensive tasks while isolating less intensive tasks on smaller teams.

5.3.6 Best Practices

1. **Define Membership Carefully:** Ensure logical partitioning of images for tasks and avoid excessive inter-team communication.
2. **Use Descriptive Team Names:** Clearly name `TEAM_TYPE` variables for readability and maintenance.
3. **Combine with Coarray Features:** Use coarrays for data sharing across teams and leverage collective subroutines such as `CO_SUM` and `CO_MAX`.
4. **Debug with Team Scopes:** Print image indices and team sizes during initialization for verification.

Image teams in coarrays are a key feature for building scalable, hierarchical parallel programs in modern Fortran. They enable effective utilization of computational resources for complex simulations, making them an indispensable tool for developers.

6 Conclusion

SwarmStorm demonstrates the power of modern Fortran for scientific simulations, combining object-oriented programming with coarray parallelization. The modular design of the 'satellite' type and the seamless integration of coarrays make it a valuable tool for satellite interaction studies.

7 Fortran Readings

The best reference for Fortran is the *Modern Fortran Explained* series [4] by Metcalf, Reid, Cohen, and Bader. This is a living document writing by members of the Fortran Standards committee and evolves in lock step with the language.

The next two books demonstrate a careful attention to teaching through example. Clerman and Spector's book *Modern fortran: building efficient parallel applications* [1] is written around a didactic example of a tsunami simulation. Curic's book *Modern fortran: building efficient parallel applications* [2] is a more typical approach use a broad array of examples.

The book by Hanson and Tompkins, *Numerical computing with modern fortran* [3], is a brief book with many useful insights on the practical numerics with Fortran.

Parallel programming with co-arrays [5], Fortran 2018 with parallel programming [6]

A Proof of Principle

Below is the output from compiling and running the program, demonstrating its successful execution. The code creates an array of 3 blue satellites of type `Psamathe` and 2 red satellites of type `Despina`.

```
dantopa@Mac-mini.local:foxtrot $ ./swarmstorm
Created Psamathe Satellite 1
Satellite Parameters:
Model:              Psamathe
Color:              1
Mass (Metal):       1200.0000000000000
Mass (Fuel):        2500.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               TopaSat-1
UUID:                  BB:2D:DC:18:8C:20:4D:64:B5:31:CE:A6:D9:98:61:D3
Created Psamathe Satellite 2
Satellite Parameters:
Model:              Psamathe
Color:              1
Mass (Metal):       1200.0000000000000
Mass (Fuel):        2500.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               TopaSat-2
UUID:                  1C:B9:F3:2B:DD:8A:4D:09:9F:5D:6E:75:3F:95:51:B7
Created Psamathe Satellite 3
Satellite Parameters:
Model:              Psamathe
Color:              1
Mass (Metal):       1200.0000000000000
Mass (Fuel):        2500.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               TopaSat-3
```

```
UUID:                   8C:E2:62:72:0C:02:4E:88:BF:70:E0:F8:41:36:35:A1
Satellite Parameters:
Model:              Despina
Color:              2
Mass (Metal):       2450.0000000000000
Mass (Fuel):        3750.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               Despina-Class
UUID:                  00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
Created Despina Satellite 1
Satellite Parameters:
Model:              Despina
Color:              2
Mass (Metal):       2450.0000000000000
Mass (Fuel):        3750.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               DanSat-1
UUID:                  98:82:0D:21:A9:D0:4A:93:BF:DD:B5:4D:F2:1A:D3:8B
Created Despina Satellite 2
Satellite Parameters:
Model:              Despina
Color:              2
Mass (Metal):       2450.0000000000000
Mass (Fuel):        3750.0000000000000
Radius (Orbit):     0.0000000000000000
Energy (Kinetic):   0.0000000000000000
Time Born:          0.0000000000000000
Time End:           0.0000000000000000
Time Attack:        0.0000000000000000
Moniker:               DanSat-2
UUID:                  27:22:B1:EA:85:E6:4F:82:B3:74:F2:AD:1A:0E:6D:3D

completed at 2024-12-06 10:35:10

CPU time used =      0.267E-02 seconds.

compiler version: GCC version 14.2.0.

compiler options: -fdiagnostics-color=auto -fPIC
    -feliminate-unused-debug-symbols -mmacosx-version-min=15.0.0 -mtune=core2 -g
    -Og -Wpedantic -Wall -Wextra -Waliasing -Wsurprising -Wimplicit-procedure
    -Wintrinsics-std -Wfunction-elimination -Wc-binding-type -Wrealloc-lhs-all
    -Wuse-without-only -Wconversion-extra -fno-realloc-lhs
    -ffpe-trap=denormal,invalid,zero,overflow,underflow -fbacktrace
    -fmax-errors=10 -fcheck=all -fcheck=pointer -fno-protect-parens
    -faggressive-function-elimination -finit-derived -frecursive -J obj.

STOP Successful termination for "swarmstorm.f08".
```

References

1 Norman S Clerman and Walter Spector. *Modern Fortran: style and usage*. Cambridge University Press, 2011.

2 Milan Curcic. *Modern Fortran: Building efficient parallel applications*. Manning Publications, 2020.

3 Richard J. Hanson and Tim Hopkins. *Numerical Computing with Modern Fortran*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2013. DOI: 10.1137/1.9781611973129. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973129. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973129.

4 Michael Metcalf et al. *Modern Fortran Explained: Incorporating Fortran 2023*. Oxford University Press, 2024.

5 Robert W Numrich. *Parallel programming with co-arrays*. Chapman and Hall/CRC, 2018.

6 Subrata Ray. *Fortran 2018 with parallel programming*. Chapman and Hall/CRC, 2019.