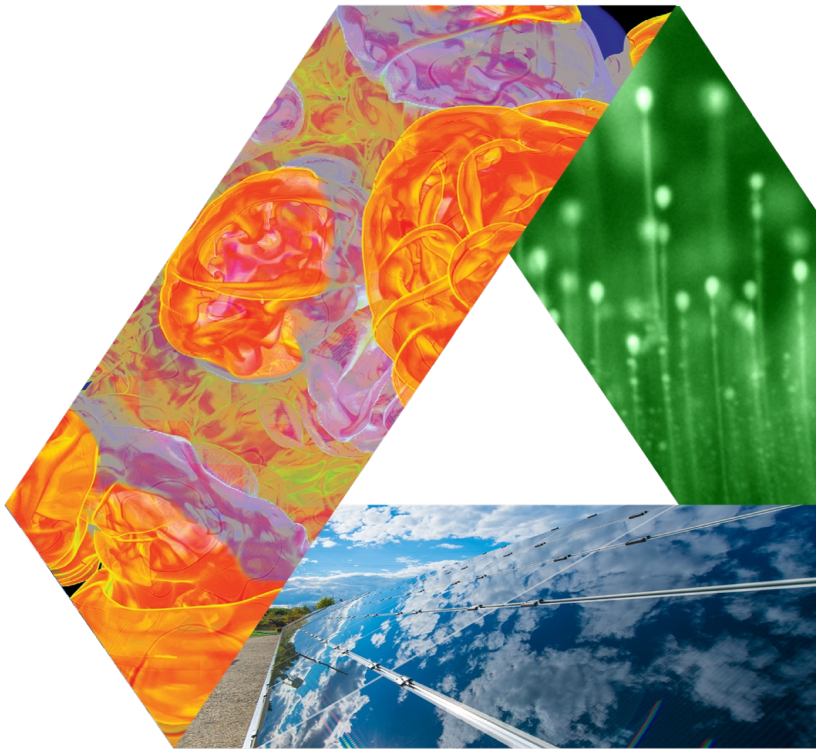
IDEAS productivity

Portability of Composable Multiphysics Applications



Anshu Dubey

Contributors: Klaus Weide, Jared O'Neal, Fromm, Mohamed Wahib, Tom Klosterman, Youngjun Lee, Wesley kweinciski

Argonne National Laboratory

- June 27, 2024

ECP EXASCALE COMPUTING PROJECT

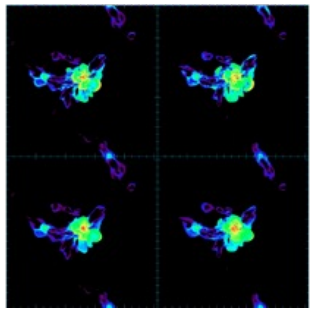
U.S. DEPARTMENT OF ENERGY | Office of Science

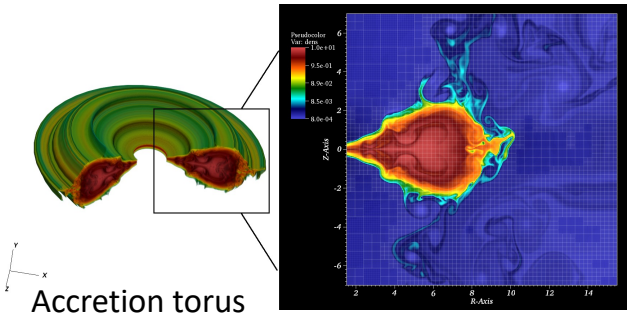
acknowledgements

- This work was partly supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Flash-X: A composable Multiphysics Software

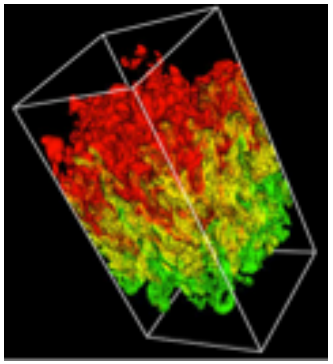
Cosmological cluster formation



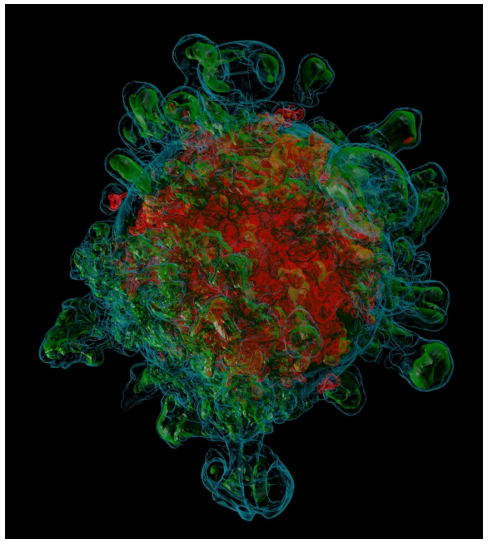


Accretion torus

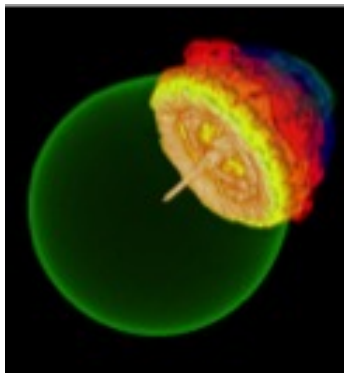
Rayleigh-Taylor instability



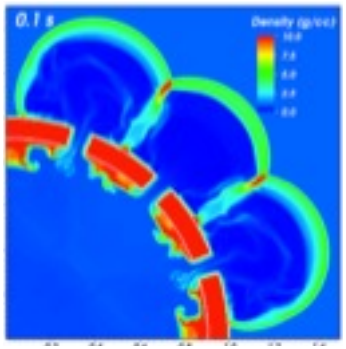
Core Collapse Supernova



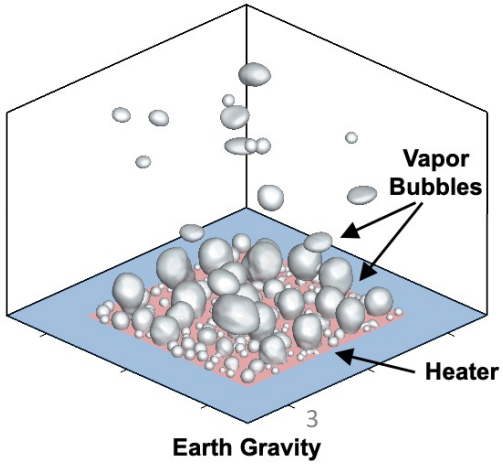
Type Ia SN



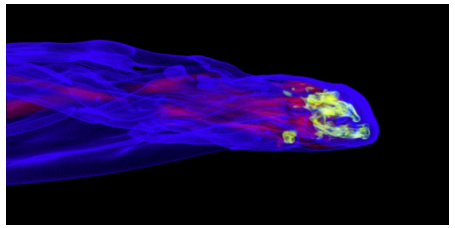
Rigid body structure



Gravity impact on boiling



Ram pressure stripping



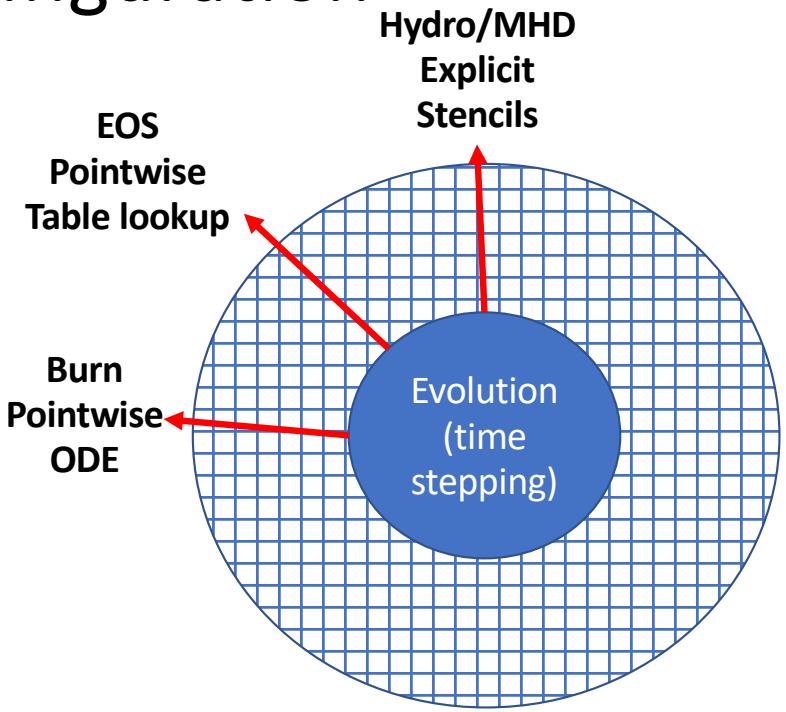
Flash-X fundamentals

- Base discretization is Eulerian, using finite volume methods
- Fundamental abstraction is a block with surrounding halo of ghost cells
- Adaptive mesh refinement is used to reduce both memory footprint and computations
- Lagrangian framework super-imposed on Eulerian discretization
  - Used as tracers, N-body problems, fluid-structure interactions

Extensible Software Architecture

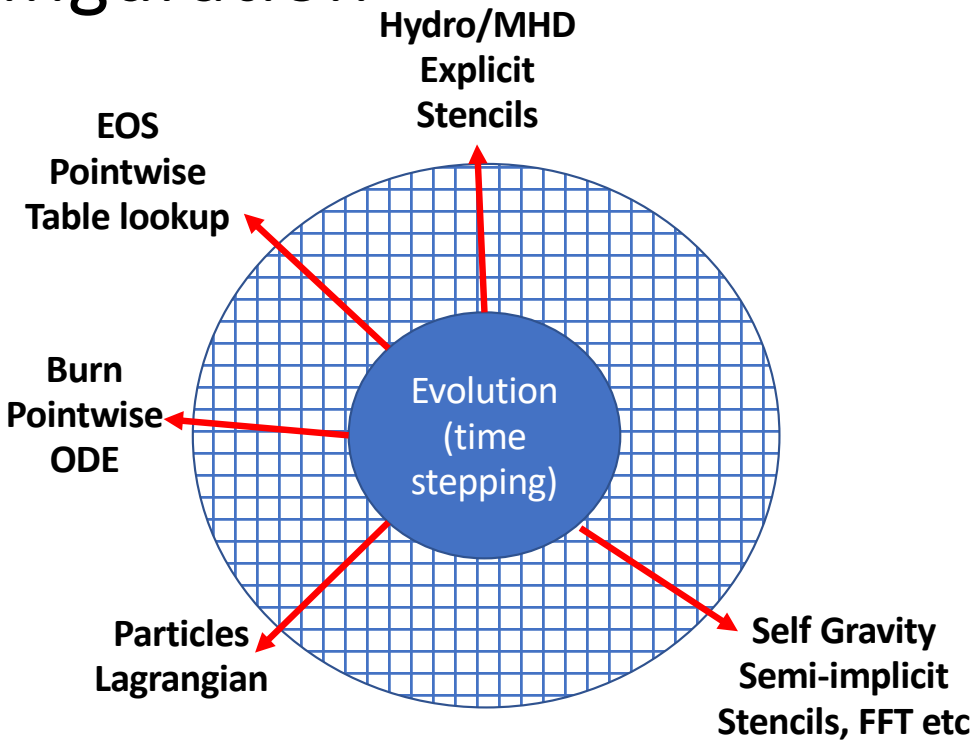
- Building blocks of code
  - Hierarchy of granularity
  - Units, subunits, components
- Multiple alternative implementations
  - Null implementations of API
  - High degree of composability
  - High degree of customizability
- A tool that can arbitrate on what to include when
  - Self describing code components

Flash-X Configuration



Cellular

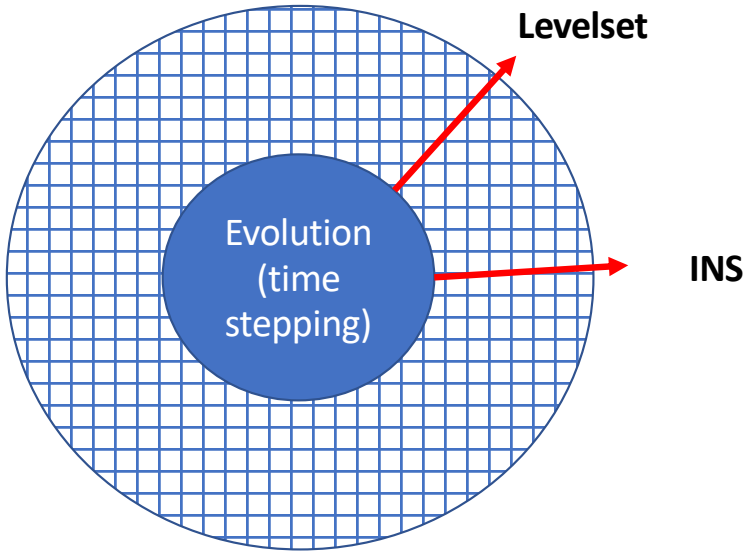
Flash-X Configuration



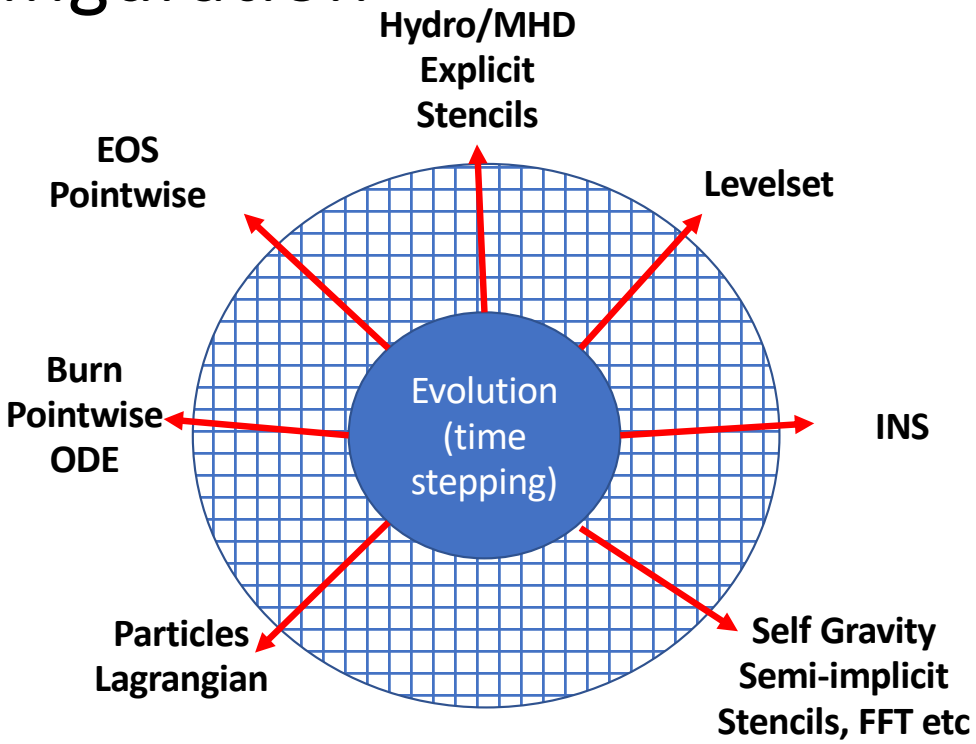
Supernova

Flash-X Configuration

Boiling

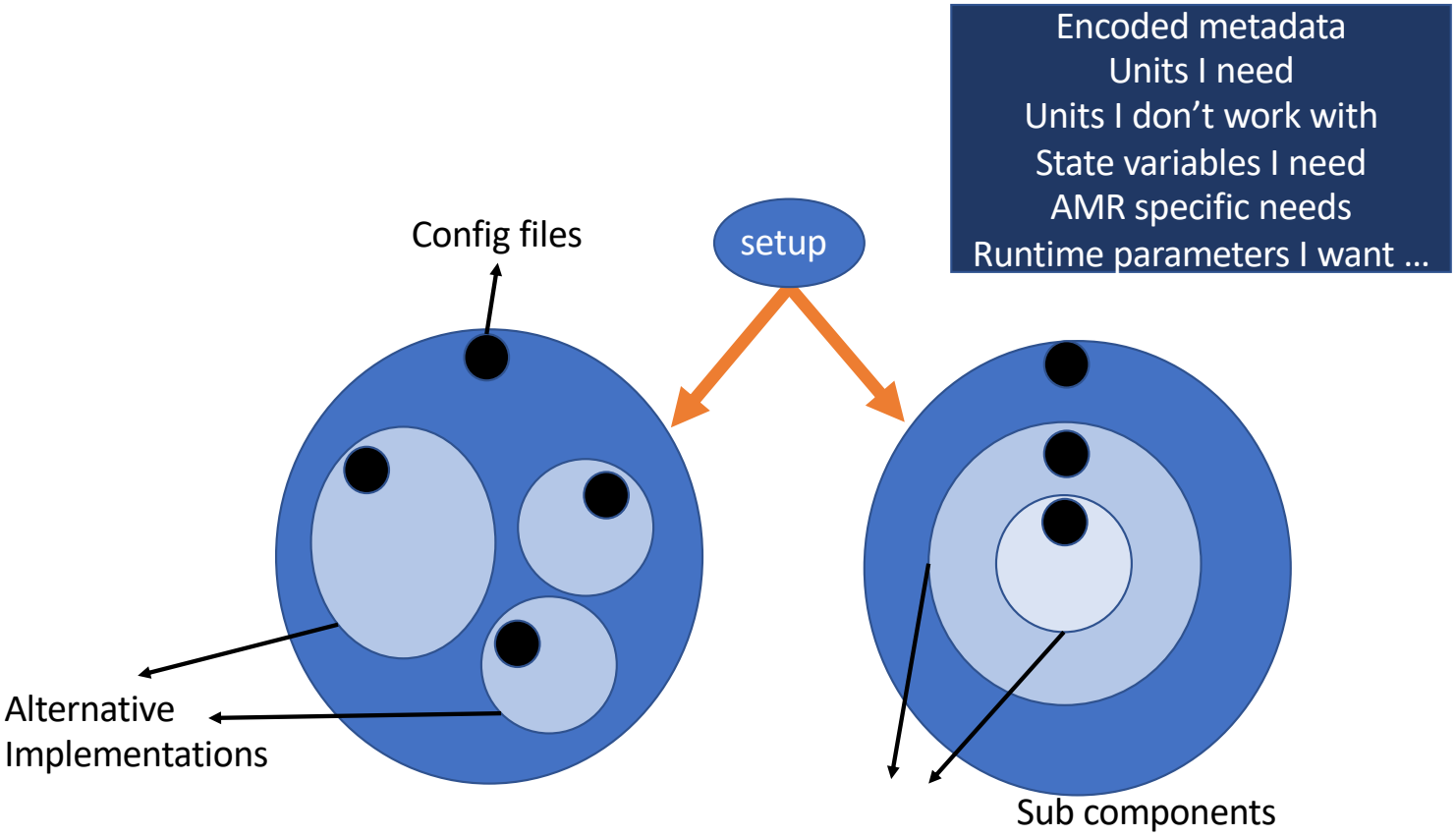


Flash-X Configuration



Anshu Dubey, Katie Antypas, Murali K. Ganapathy, Lynn B. Reid, Katherine Riley, Dan Sheeler, Andrew Siegel, Klaus Weide, Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code, Parallel Computing, Volume 35, Issues 10–11, 2009, Pages 512-522, https://doi.org/10.1016/j.parco.2009.08.001.

The Key to Composability



```
#       Config file for the gravity module.  Available sub-modules:

#         Constant    Spatially/temporally constant gravitational field
#         PlanePar    1/r^2 field for a distant point source
#         PointMass   1/r^2 field for an arbitrarily placed point source
#         Poisson     Field for a self-gravitating matter distribution
#   UserDefined A user-defined field

REQUIRES Driver
DEFAULT Constant
PPDEFINE GRAVITY
EXCLUSIVE Constant PlanePar PointMass Poisson UserDefined
PARAMETER useGravity BOOLEAN TRUE
```

Rearchitect for heterogeneity

- Switching over to C++ short-sighted
  - Popular performance portability solutions are useful for this generation
  - Platforms appear poised for more heterogeneity
    - Tiles, specialized accelerators etc
  - Even three GPU variants require huge teams to maintain abstraction
  - On-ramping possibilities are nearly non-existent
- Newer languages hold more promise
  - More robust abstractions for expressing data locality => more portability
  - Scientific software needs are first class objects
  - Not all perform as well, but they likely will
- Opportunity to devise solutions that work and don't preclude transition to other languages

Philosophy of Design

- Let the code developer decide what should be done for optimization on a platform
  - Make it easy to have that happen without coding to metal
  - Keep language specific features to a minimum
- Tools are execution engines – little to no inferencing
  - Simple and easy to maintain by non-experts
  - Human-in-the-loop handles inferencing and corner cases
  - Leave open possibility of plugging in third party inferencing engines

Mechanisms Needed by the Code

Mechanisms to unify expression of computation
- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to move work and data to computational targets
- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data offnode

Mechanisms to map work to computational targets
- Figuring out the map
  - Expression of dependencies
  - Cost models
- Expressing the map

So what do we need?

- Abstractions layers
- Code transformation tools
- Data movement orchestrators

Mechanisms Needed by the Code

Mechanisms to unify expression of computation

- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Express code with embedded macros

- Let macros have multiple alternative definitions
- Implement mechanism to select specific macro definition
- Implement mechanism to safely include more than one definition
- Allow inline, recursion and arguments in macros

Mechanisms Needed by the Code

Version 1

```
integer,parameter:: N=10, M=12
integer:: i,j
real,dimension(N,M) :: A,B
real,dimension(N) :: temp

do j=1,n
   do i=1,m
      temp(i)=A(i,j)*0.5
      A(i,j) = A(i,j)*B(i,j)
   end do
   do i=1,m
      B(i,j)=temp(i)+A(i,j)
   end do
end do
```

Version 2

```
integer,parameter:: N=10, M=12
integer:: i,j
real,dimension(N,M) :: A,B
real :: temp


do j=1,n
   do i=1,m
      temp=A(i,j)*0.5
      A(i,j) = A(i,j)*B(i,j)
      B(i,j)=temp+A(i,j)
   end do
end do
```

Unified Code

```
integer,parameter:: N=10, M=12
  integer:: i,j
  real,dimension(N,M) :: A,B
  @M declare

   do j=1,n
     do i=1,m
        @M tmp =A(i,j)*0.5
        A(i,j) = A(i,j)*B(i,j)
     @M break_loop
        B(i,j) = @M tmp +A(i,j)
     end do
 end do
```

Mechanisms Needed by the Code

Unified Code

```
integer,parameter:: N=10, M=12
  integer:: i,j
  real,dimension(N,M) :: A,B
  @M declare

   do j=1,n
     do i=1,m
        @M tmp =A(i,j)*0.5
        A(i,j) = A(i,j)*B(i,j)
     @M break_loop
        B(i,j) = @M tmp +A(i,j)
     end do
 end do
```

```
[declare]
definition =
  real,dimension(N) :: temp

[tmp]
definition =
  temp(i)

[break_loop]
definition =
  end do
  do i=1,m
```

Version 1

```
  integer,parameter:: N=10, M=12
  integer:: i,j
  real,dimension(N,M) :: A,B
  real,dimension(N) :: temp

  do j=1,n
     do i=1,m
        temp(i)=A(i,j)*0.5
        A(i,j) = A(i,j)*B(i,j)
     end do
     do i=1,m
        B(i,j)=temp(i)+A(i,j)
     end do
end do
```

Mechanisms Needed by the Code

Unified Code

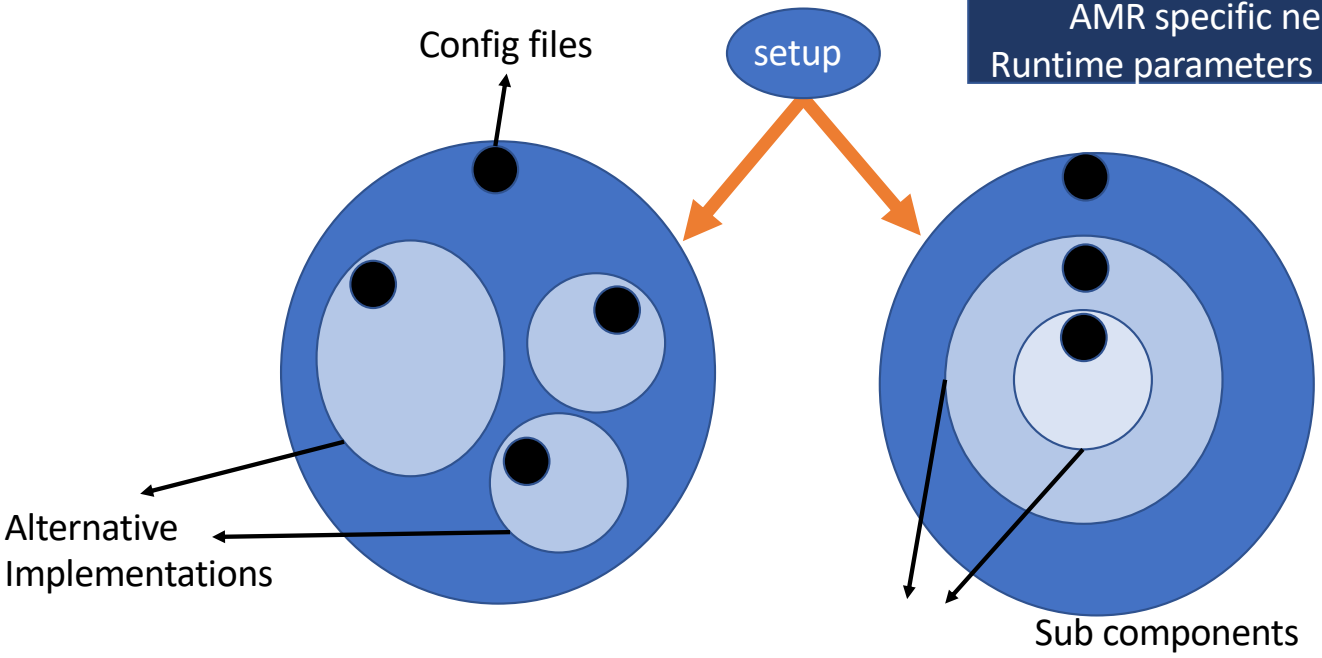
```
integer,parameter:: N=10, M=12
  integer:: i,j
  real,dimension(N,M) :: A,B
  @M declare

   do j=1,n
     do i=1,m
        @M tmp =A(i,j)*0.5
        A(i,j) = A(i,j)*B(i,j)
     @M break_loop
        B(i,j) = @M tmp +A(i,j)
     end do
 end do
```

```
[declare]
definition =
  real :: temp

[tmp]
definition =
  temp

[break_loop]
definition =
```

Version 2

```
integer,parameter:: N=10, M=12
integer:: i,j
real,dimension(N,M) :: A,B
real :: temp


do j=1,n
   do i=1,m
      temp=A(i,j)*0.5
      A(i,j) = A(i,j)*B(i,j)
      B(i,j)=temp+A(i,j)
   end do
end do
```

The Key to Composability



Encoded metadata
Units I need
Units I don't work with
State variables I need
AMR specific needs
Runtime parameters I want …

Lowest granularity -- subroutine

Express code with embedded macros

- Let macros have multiple alternative definitions
- Implement mechanism to select specific macro definition
- Implement mechanism to safely include more than one definition
- Allow inline, recursion and arguments in macros

CPU Specific Definitions

[indices] definition =

[loop] definition =

[loop_end] definition =

GPU Specific Definitions

[indices]
definition =
,i1,i2,i3

[loop]
definition=
do i3 = i3LOW, i3HIGH
do i2 = i2LOW,i2HIGH
do i1 = i1LOW,i1HIGH

[loop_end]
definition =
enddo
enddo
enddo

Common Definition

[hy_fluxes]
definition =
@M loop
if (flux(1@M indices) > 0.) then
flux(:@M indices) = XL(:@M indices)
else
flux(:@M indices) = XR(:@M indices)
end if
@M loop_end

Source code using macro

@M hy_fluxes

Translated Code for GPU

```
do i3 = i3LOW, i3HIGH
      do i2 = i2LOW,i2HIGH
         do i1 = i1LOW,i1HIGH
           if (flux(1 ,i1,i2,i3) > 0.) then
             flux(: ,i1,i2,i3) =  uPlus(: ,i1,i2,i3)
          else
            flux(: ,i1,i2,i3) =  uMinus(: ,i1,i2,i3)
          end if
         enddo
    enddo
enddo
```

Translated Code for CPU

```
if (flux(1) > 0.) then
      flux(:) =  uPlus(:)
  else
    flux(:) =  uMinus(:)
  end if
```

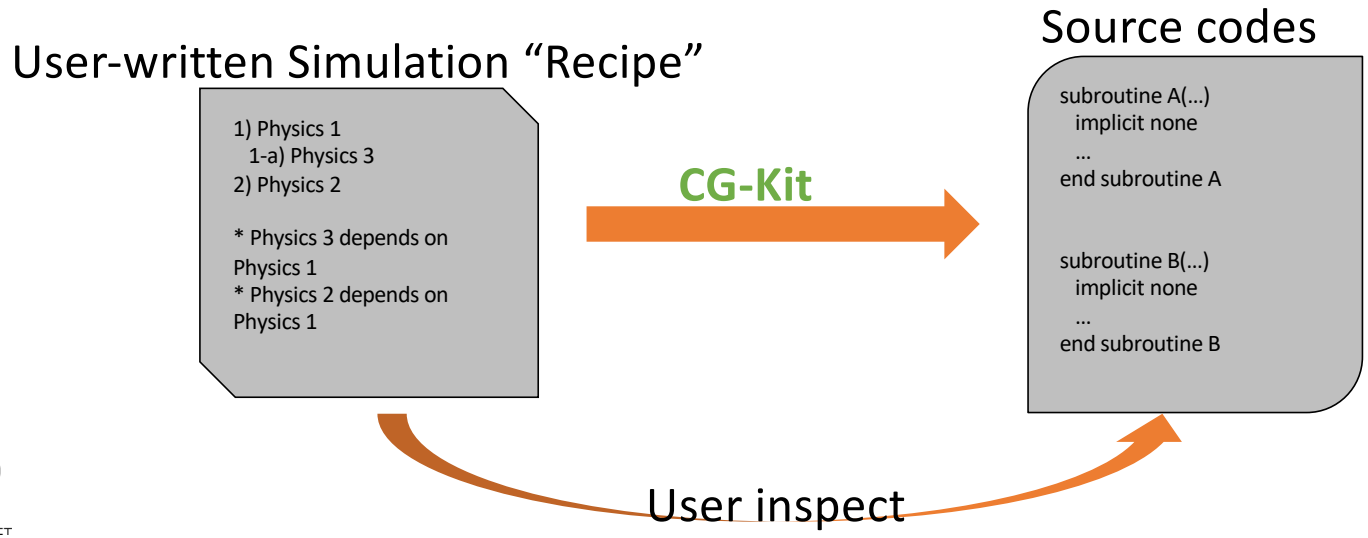
```
!  Begin loop over zones
@M hy_DIR_HOST_parallel_if_threaded &
@M hy_DIR_HOST_getFaceFlux_shared &
@M hy_DIR_HOST_getFaceFlux_private
do dir = 1, NDIM
   ! call Timers_start("prepare_scratch")
   @M hy_DIR_TARGET_update_to([lim, limgc, klim, dir, stage])

   hy_flux(1:NFLUXES,@M hy_bounds(limgc))=>hya_flux
   hy_flat(@M hy_bounds(limgc))=>hya_flat
   hy_shck(@M hy_bounds(limgc))=>hya_shck
   hy_grv(@M hy_bounds(limgc))=>hya_grv

   hy_rope(1:NRECON,@M hy_bounds(limgc)) => hya_rope
   hy_uPlus(1:NRECON,@M hy_bounds(limgc)) => hya_uPlus
   hy_uMinus(1:NRECON,@M hy_bounds(limgc)) => hya_uMinus

   @M hy_DIR_TARGET_enter_data(alloc, [hy_flux, hy_flat, hy_shck, hy_grv, hy_rope, hy_uPlus, hy_uMinus])
   ! call Timers_stop("prepare_scratch")

   ! call Timers_start("setRope")
   @M hy_DIR_HOST_do_collapse(2)
   @M hy_DIR_TARGET_parallel_loop(3) &
   @M hy_DIR_TARGET_setRope_private &
   @M hy_DIR_TARGET_setRope_shared
   @M hy_permute_loop(limgc, lim)
```

Mechanisms Needed by the Code

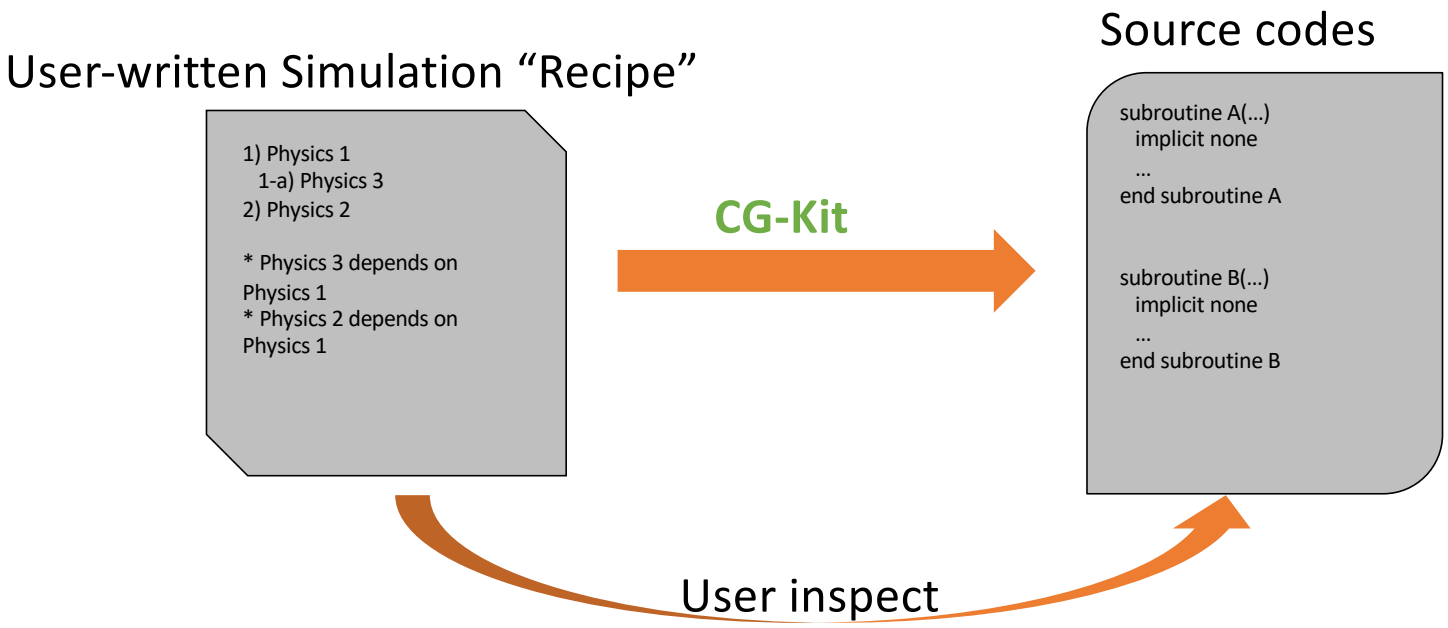
Mechanisms to map work to computational targets

- Figuring out the map
  - Expression of dependencies
  - Cost models
- Expressing the map

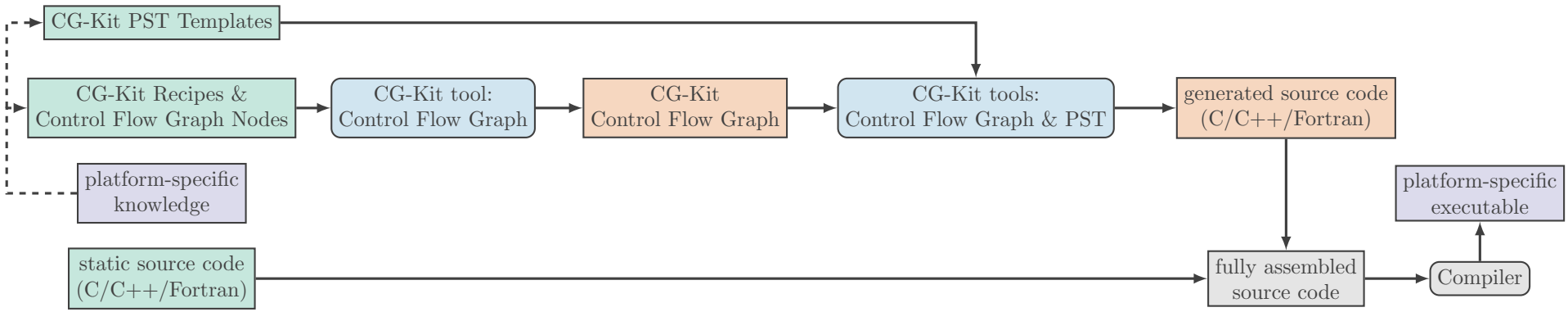
CGKit : Generating Code from Recipes and code Templates



Flash-X: Multiphysics simulation code

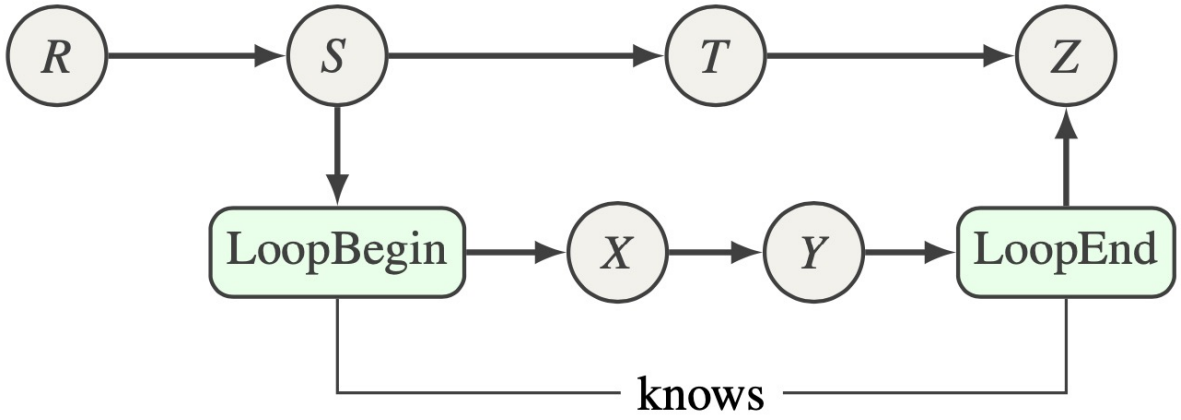


CG-Kit


CG-Kit PST Templates
CG-Kit Recipes & Control Flow Graph Nodes
CG-Kit tool: Control Flow Graph
CG-Kit Control Flow Graph
CG-Kit tools: Control Flow Graph & PST
generated source code (C/C++/Fortran)
platform-specific knowledge
platform-specific executable
static source code (C/C++/Fortran)
fully assembled source code
Compiler

Control Flow Graph

- CG-Kit generates Control Flow Graph by reading user input (recipe).
  - Represented as a directed acyclic graph (DAG)
  - Each node represents user-defined code generation operations.
    - PST
  - Each edge describes dependencies between nodes.



Variants For Hydrodynamics

- Flash-X supports two block-structured AMR grid backends
  - Paramesh: Octree-based
  - AMReX: Level-based
- Each has different preferences for flux correction at fine-coarse boundaries

```
do all_blocks
  ! hydrodynamics updates
end do
call communicate_fluxes() ! p2p communication
do all_blocks
  ! flux correction
end do
```

Paramesh

```
do lev = max_level, 1, -1
  call communicate_fluxes() ! p2p communication
  do blocks_on(level = lev)
    ! hydrodynamics updates
    ! flux correction
  end do
end do
```

AMReX

Spark Variants (RK Modes)

- The performance trade-offs from Telescoping mode can be varying depending on the simulation size, computation intensity, and hardware
- The best practice would be to conduct the performance analysis ahead of production simulations

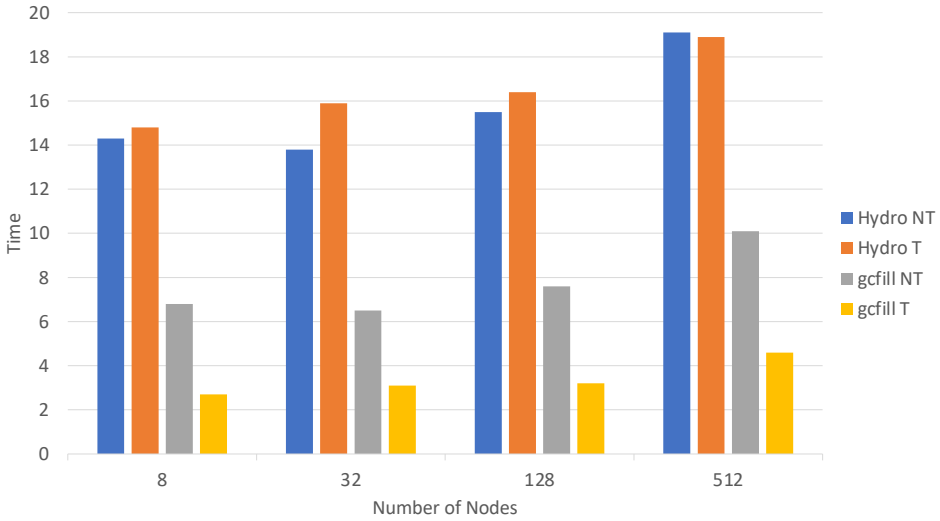
```
do stage = 1, max_stage
  call fill_guardcells() ! p2p communication
  do all_blocks
    ! block initializations
    ! intra stage calculations
  end do
end do
```

Non-telescoping (traditional)

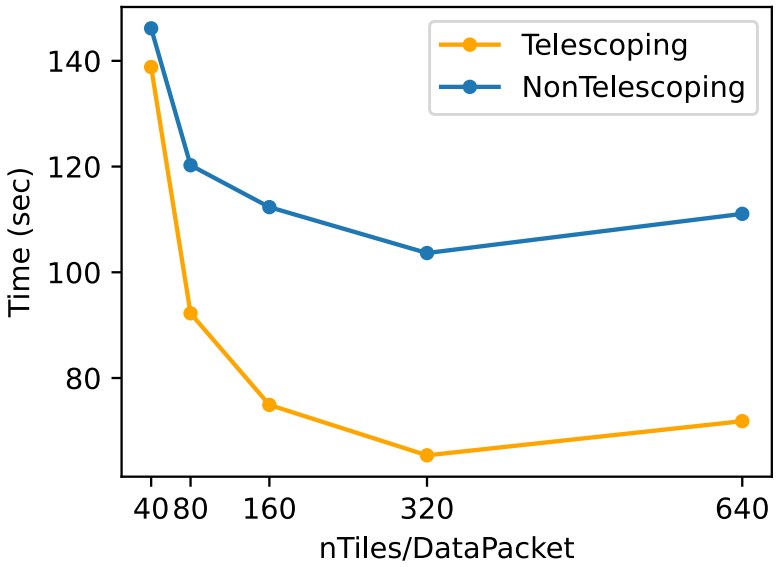
```
call fill_guardcells() ! p2p communication
do all_blocks
  ! block initializations
  do stage = 1, max_stage
    ! intra stage calculations
  end do
end do
```

Telescoping

Spark Variants (RK Modes)



Performance comparisons on the CPU-only machine (Fugaku)



Performance comparisons on the GPU (Nvidia RTX 3080)

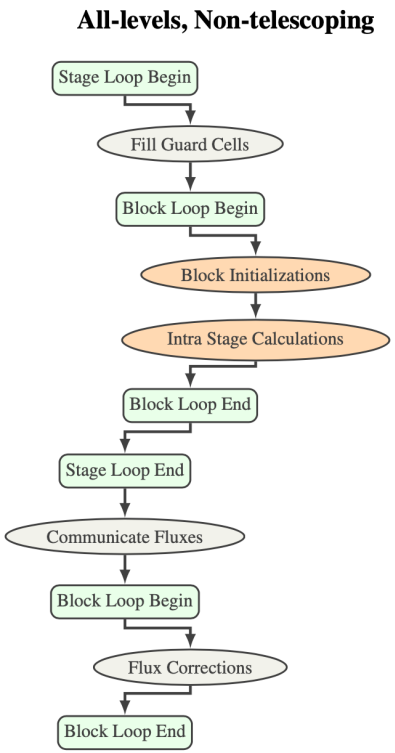
Spark Variants

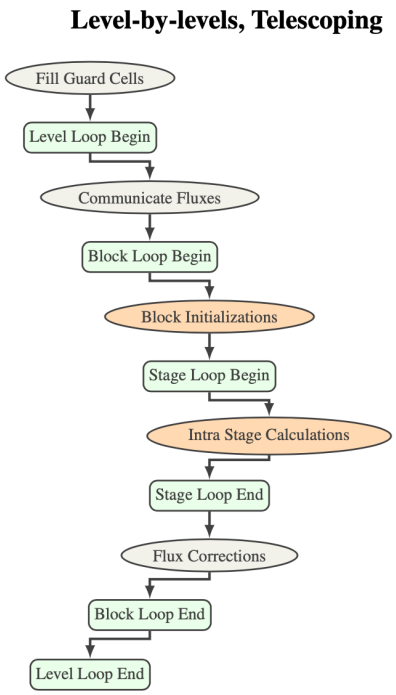
Algorithm 4.7 Block initializations for Spark

```
1:  function SPARK_BLOCK_INIT(recipe, root, nodes)
2:      n = nodes
3:      shockDet ← recipe.add(n.shockDet)(root)
4:      initSoln ← recipe.add(n.initSoln)(root)
5:      end ← recipe.add(n.null)([shockDet, initSoln])
        ▷ "null" node, blocking multiedge, does nothing for PST.
6:      return end
7:  end function
```

Algorithm 4.8 Intra stage calculations for Spark

```
1:  function SPARK_INTRA_STAGE(recipe, root, nodes)
2:      n = nodes
3:      grvAccel ← recipe.add(n.grvAccel)(root)
4:      calcLims ← recipe.add(n.calcLims)(grvAccel)
5:      calcFlux ← recipe.add(n.calcFlux)(calcLims)
6:      fluxBuff ← recipe.add(n.fluxBuff)(calcFlux)
7:      updSoln ← recipe.add(n.updSoln)(calcFlux)
8:      calcEos ← recipe.add(n.calcEos)(updSoln)
9:      end ← recipe.add(n.null)([fluxBuff, calcEos])
        ▷ "null" node, blocking multiedge, does nothing for PST.
10:     return end
11: end function
```





Mechanisms Needed by the Code

Mechanisms to move work and data to computational targets

- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data offnode

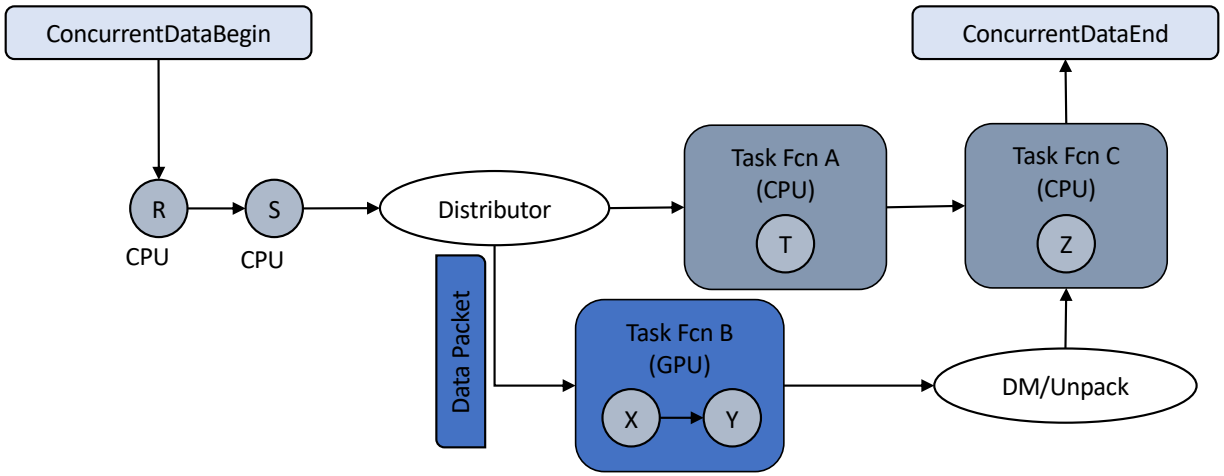
Data type

Milhoja: An extended finite state machine to move data and computation

Milhoja

- A mechanism to move data and computation between devices

Control flow graph (With Milhoja)



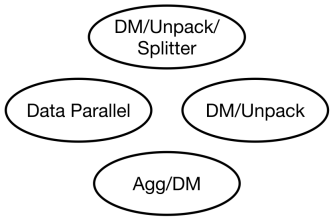
Milhoja - An extended finite state machine to move data and computation

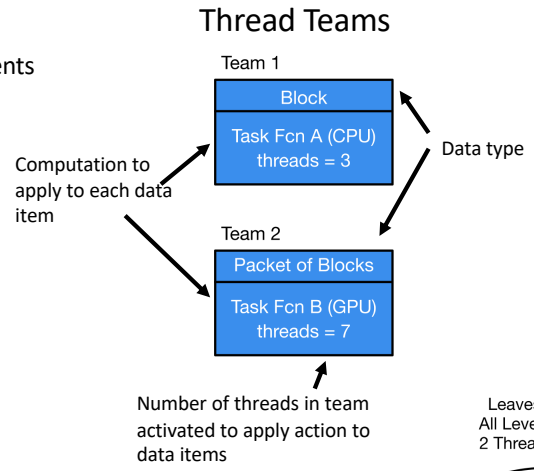
Distributors

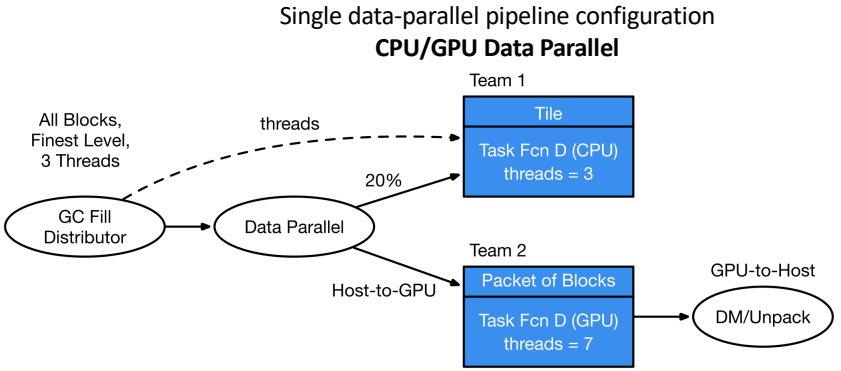
- Use block iterator
- Aggregate blocks if necessary
- Initiate asynchronous transfers if necessary
- Push blocks to other elements

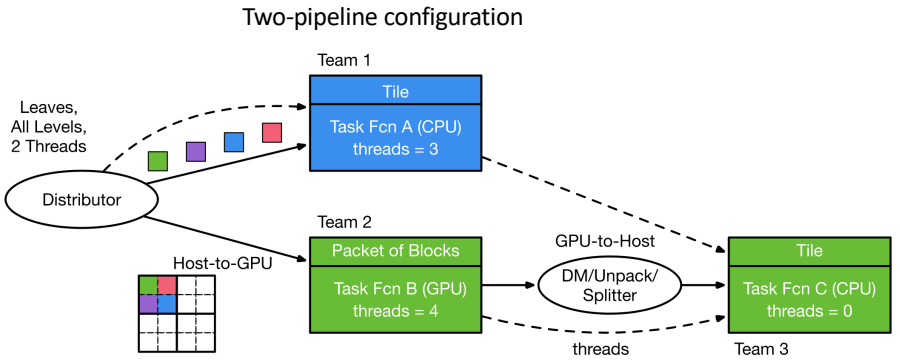
Helpers

- Initiate asynchronous transfers if necessary
- Translate data types







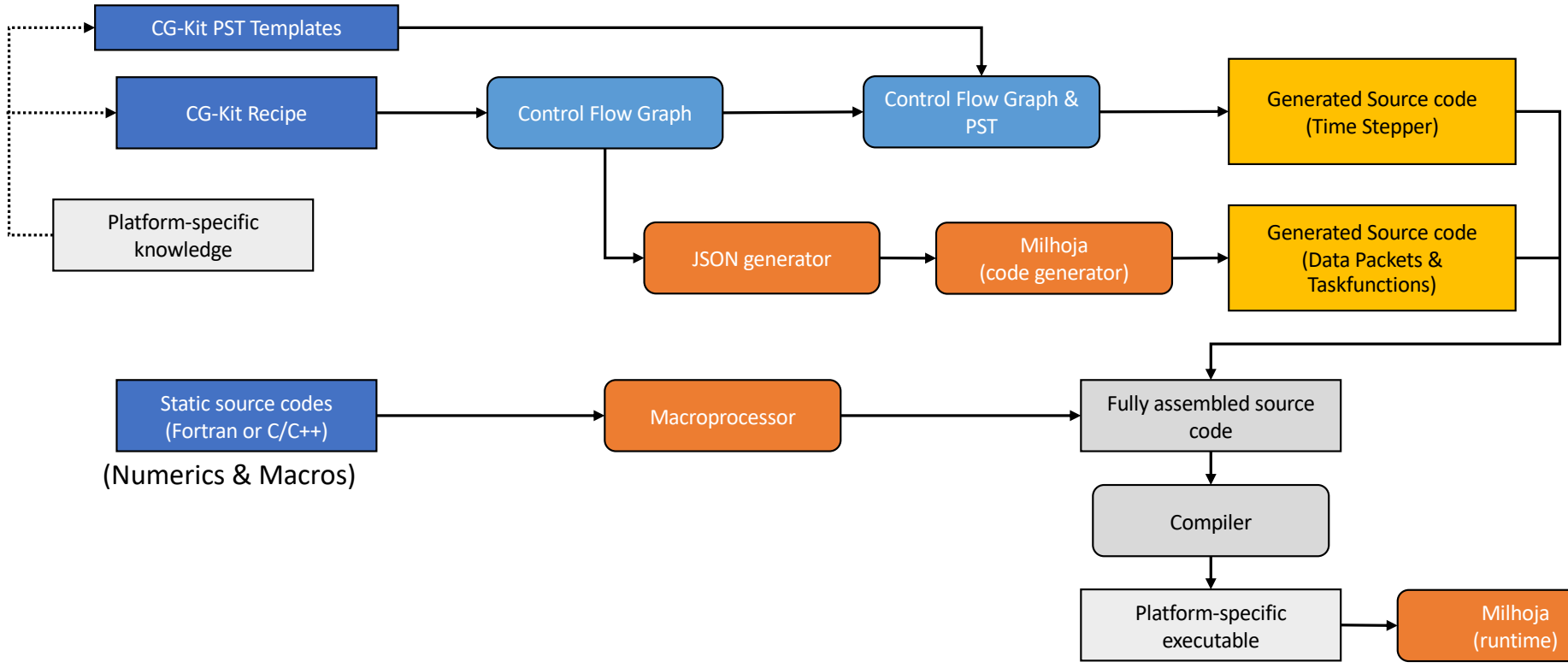


Code Generators

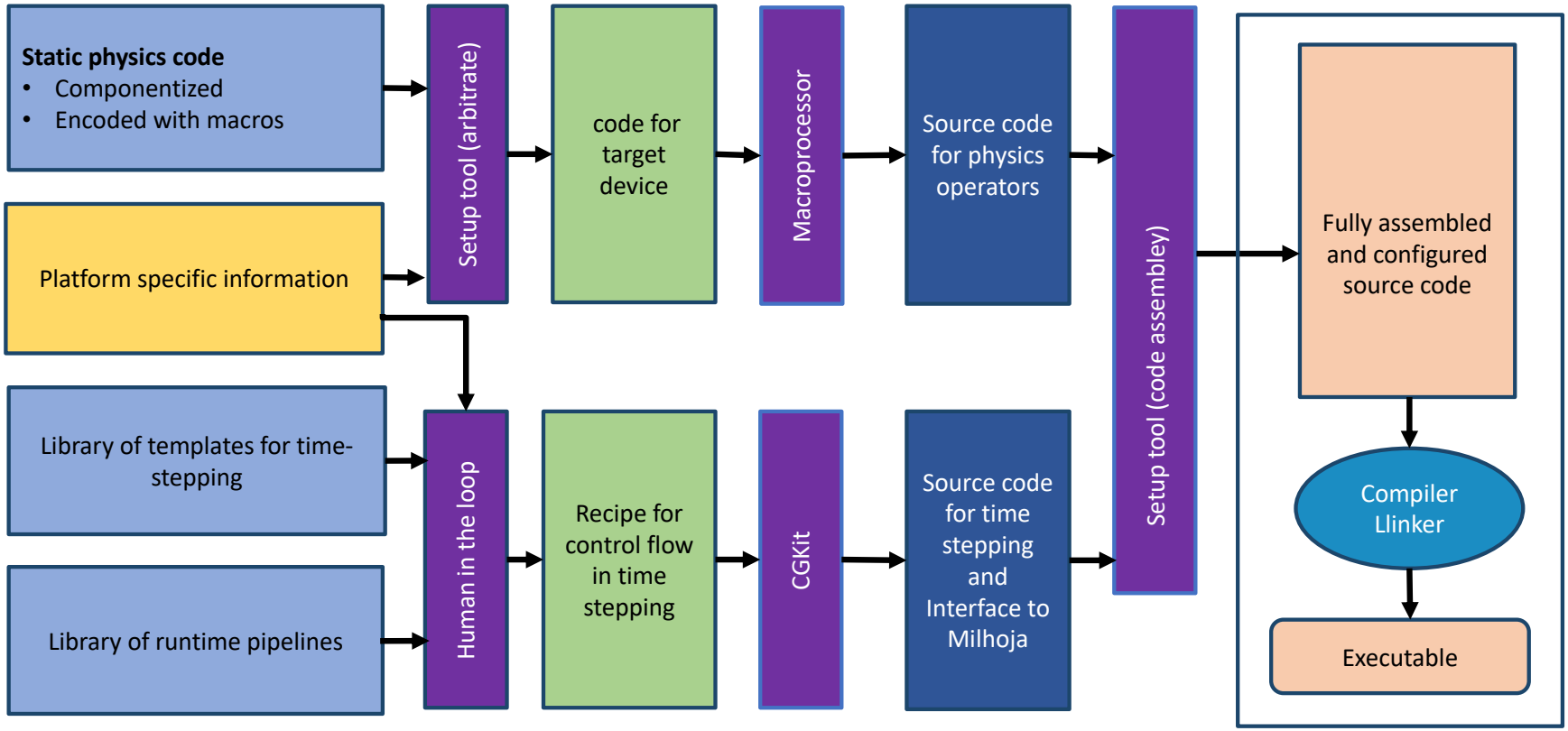
- Two Classes
  - Data packet generators
    - Parse the interface files
    - Collect all data to be put into a data packet
    - Generate code that will flatten all data into data packets
  - Task function generators
    - Consolidate functions to be invoked
    - Bookended by internode communication
    - Unpack data packets
- Decorate interface definitions with needed metadata

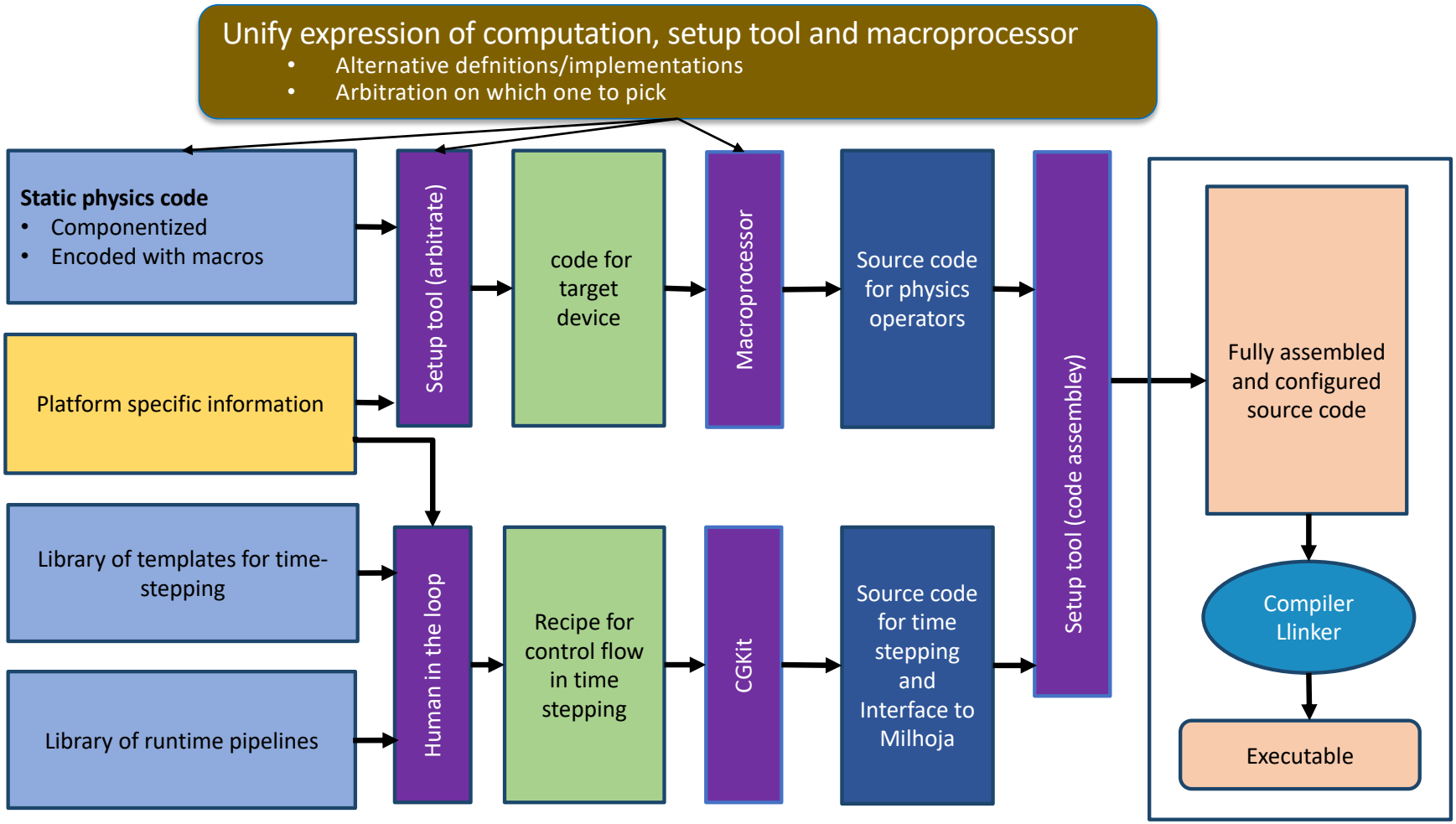
  Example

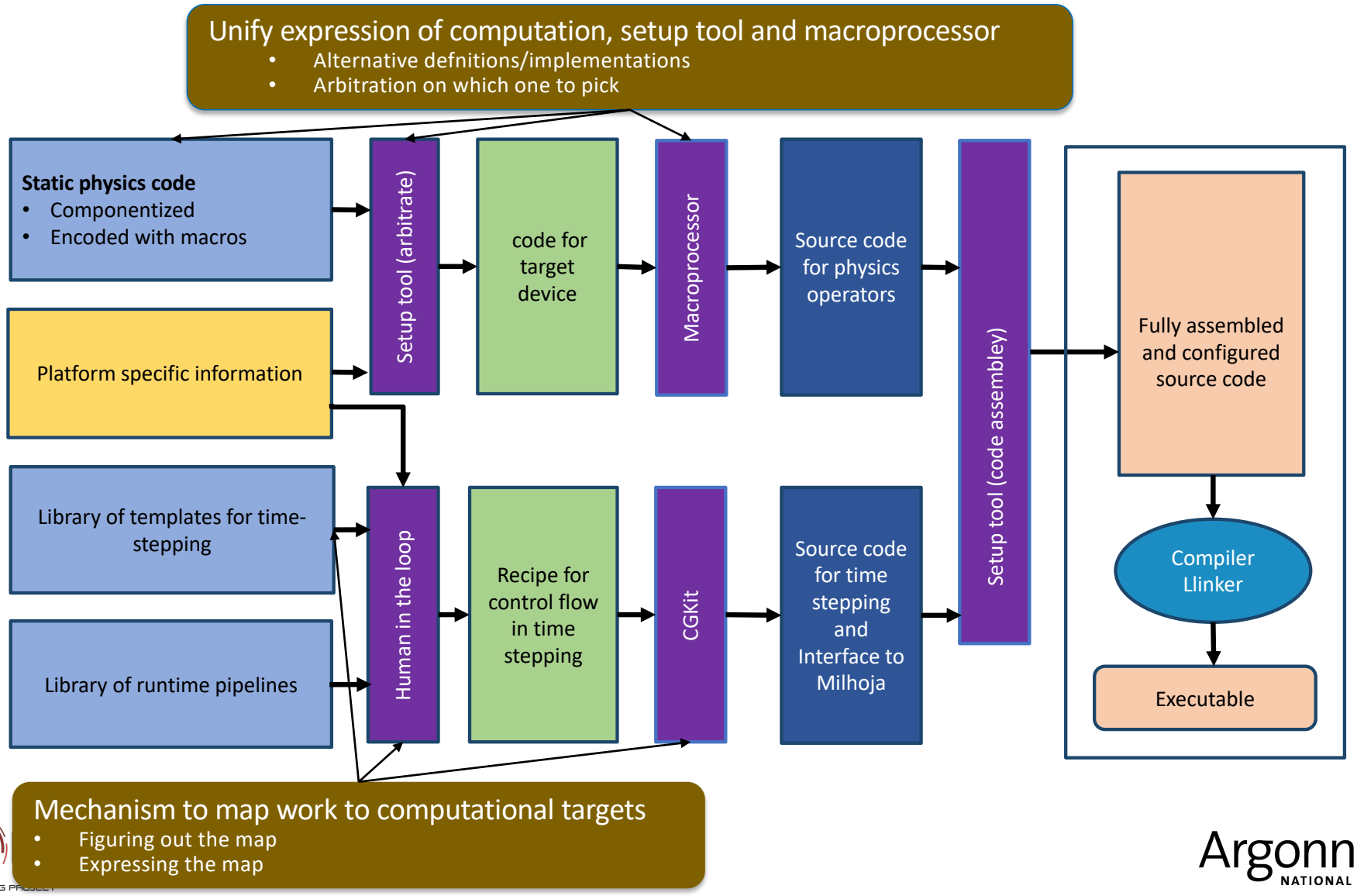
Code generation



Code Assembly

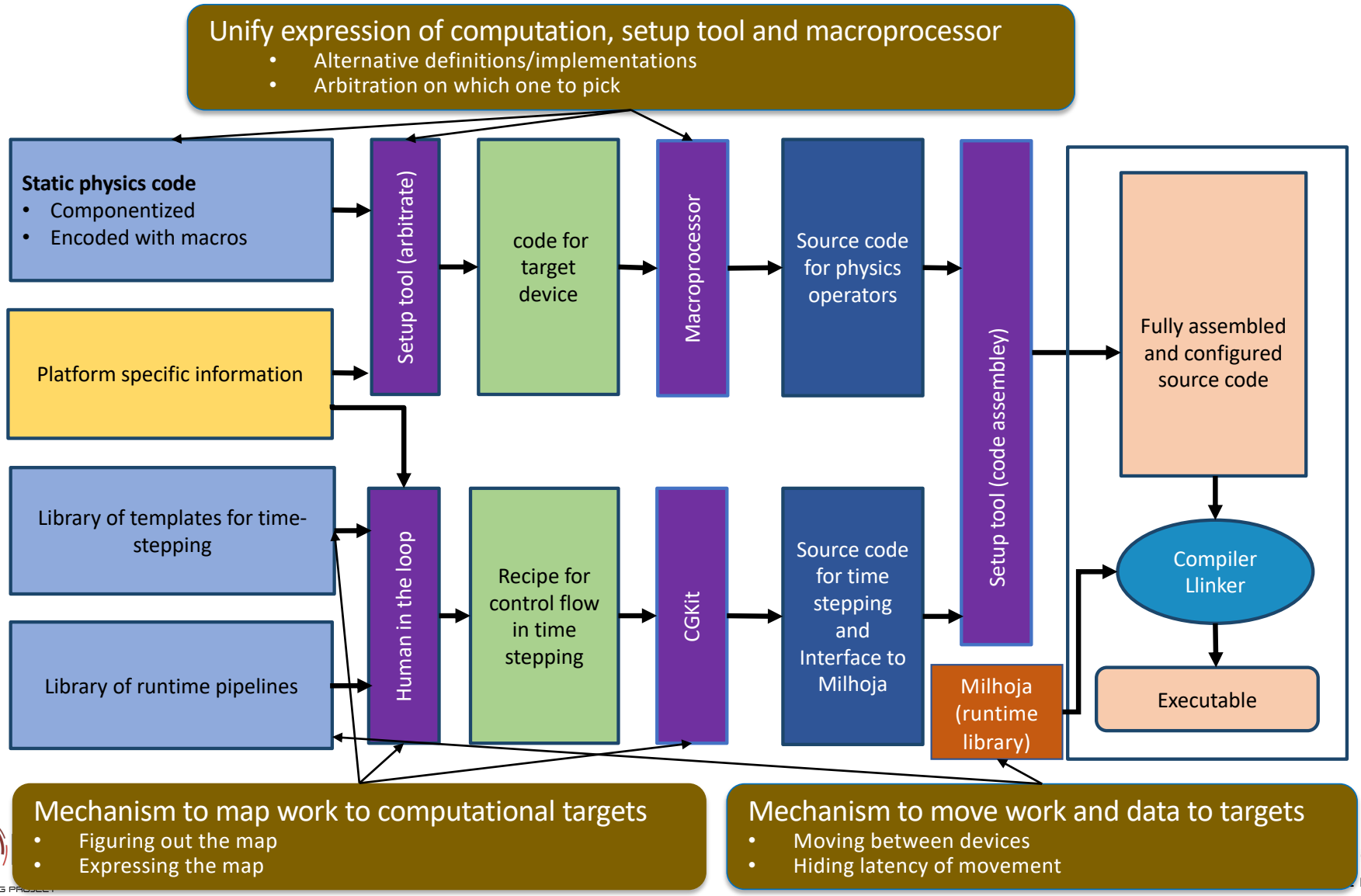



Unify expression of computation, setup tool and macroprocessor
• Alternative defnitions/implementations
• Arbitration on which one to pick
Static physics code
• Componentized
• Encoded with macros
Platform specific information
Library of templates for time-stepping
Library of runtime pipelines
Setup tool (arbitrate)
code for target device
Macroprocessor
Source code for physics operators
Human in the loop
Recipe for control flow in time stepping
CGKit
Source code for time stepping and Interface to Milhoja
Setup tool (code assembley)
Fully assembled and configured source code
Compiler Llinker
Executable



EXASCALE COMPUTING PROJECT

Argonne NATIONAL LABORATORY


Unify expression of computation, setup tool and macroprocessor
• Alternative definitions/implementations
• Arbitration on which one to pick
Static physics code
• Componentized
• Encoded with macros
Platform specific information
Library of templates for time-stepping
Library of runtime pipelines
Setup tool (arbitrate)
code for target device
Macroprocessor
Source code for physics operators
Human in the loop
Recipe for control flow in time stepping
CGKit
Source code for time stepping and Interface to Milhoja
Setup tool (code assembley)
Milhoja (runtime library)
Fully assembled and configured source code
Compiler Llinker
Executable
Mechanism to map work to computational targets
• Figuring out the map
• Expressing the map
Mechanism to move work and data to targets
• Moving between devices
• Hiding latency of movement

Porting to a new platform

- In an ideal world
  - Add to the library of runtime pipelines
  - Add to the library of recipes templates
  - Add to the knowledge base of the performance model
- In real world
  - Add variants for some solvers with alternative definitions of keys
- In the worst case
  - Develop new algorithms and add whole alternative implementation for some solvers

Questions …..