Refactoring

Anshu Dubey
Argonne National Laboratory

Software Productivity Track, ATPESC 2020

License, Citation and Acknowledgements

License and Citation

- This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0).
- **The requested citation the overall tutorial is: David E. Bernholdt, Anshu Dubey, Mark C. Miller, Katherine M. Riley, and James M. Willenbring, Software Productivity Track, in Argonne Training Program for Extreme Scale Computing (ATPESC), August 2020, online. DOI: 10.6084/m9.figshare.12719834**
- Individual modules may be cited as *Speaker, Module Title*, in Software Productivity Track…

Acknowledgements

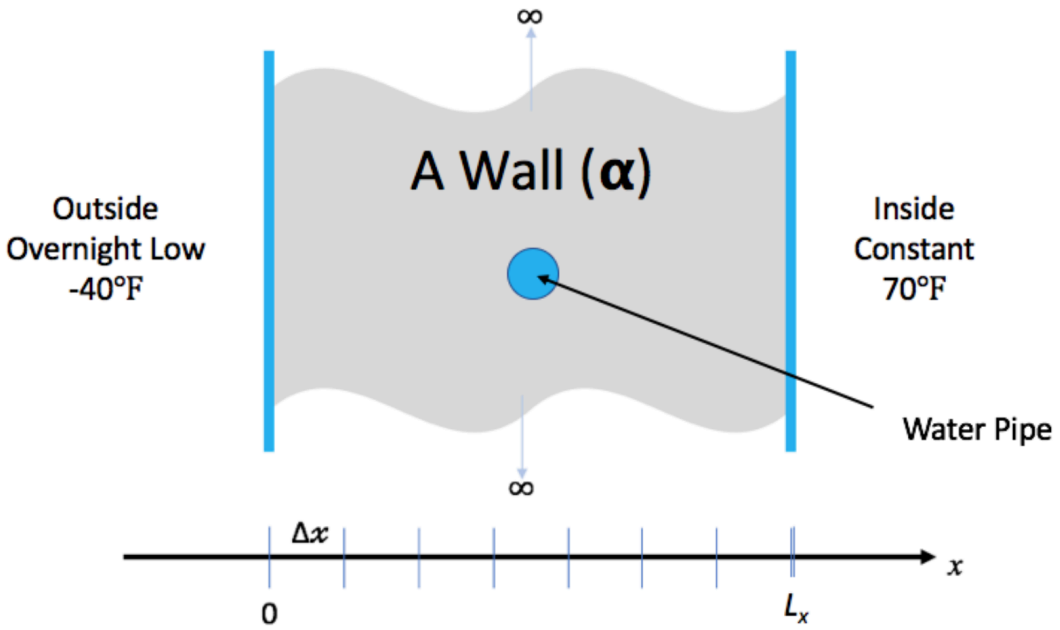
- Additional contributors include: Patricia Grubel, Rinku Gupta, Mike Heroux, Alicia Klinvex, Jared O'Neal, David Rogers, Deborah Stevens
- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Look at the Running Example

Lets say you live in a house with exterior walls made of a single material of thickness, $$L_x$$. Inside the walls are some water pipes as pictured below.



You keep the inside temperature of the house always at 70 degrees F. But, there is an overnight storm coming. The outside temperature is expected to drop to -40 degrees F for 15.5 hours. Will your pipes freeze before the storm is over?

Link to the code

https://github.com/betterscientificsoftware/hello-numerical-world-atpesc-2020/blob/main/heatAll.C

- Monolithic code
- Make it modular and more maintainable

What is Refactoring

Definition: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- Different from development
  - You have a working code
  - You know and understand the behavior
  - You have a baseline that you can use for comparison

What is Refactoring

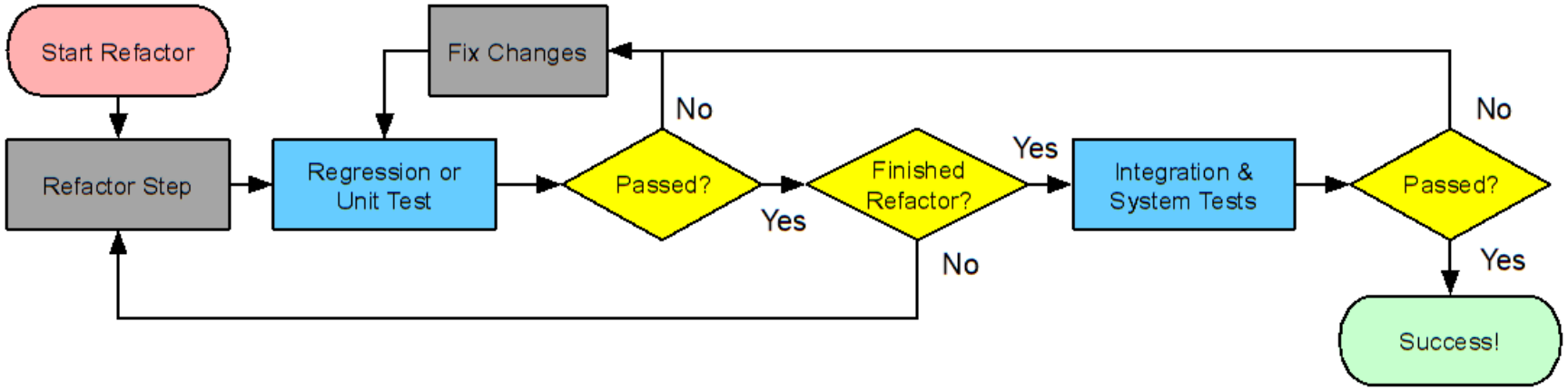
Definition: Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- Different from development
  - You have a working code
  - You know and understand the behavior
  - You have a baseline that you can use for comparison

- General motivations
  - Modularity enhancement
    - Improve sustainability
  - Release to outside users
    - Easier to use and understand
  - Port to new platforms
    - Performance portability
  - Expand capabilities
    - Structural flexibility

Refactoring

An example of workflow with testing



Considerations for Refactoring

- Know why you are refactoring
  - Is it necessary
  - Where should the code be after refactoring
- In heat example
  - It is necessary because
    - It is a monolithic code
    - No reusability of any part of the code
    - Devising tests is hard
    - Limited extensibility
  - Where do we want to be after refactoring
    - Closer to the version that you encountered in math libraries track
    - More modular, maintainable and extensible

Considerations for Refactoring

- Know the scope of refactoring
  - How deep a change
  - How much code will be affected
- In heat example
  - No capability extension
  - No performance consideration
  - Cleaner, more maintainable code

- What do we do
  - Separate out utilities, generalize interfaces
  - Separate out integration function
    - Make a general interface to allow alternative implementations
  - Create a general build function
  - No new code or intrusive changes

Before Starting

- Know your cost estimates
- Verification
  - Check for coverage provided by existing tests
  - Develop new tests where there are gaps
  - Make sure tests exist at different granularities
    - There should be demanding integration and system level tests
- Know your bounds
  - on acceptable behavior change
  - error bounds
    - bitwise reproduction of results unlikely after transition
- Map from here to there

Incorporate testing overheads into refactoring cost estimates

Cost estimation

The biggest potential pitfall

- Can be costly itself if the project is large
- Most projects do a terrible job of estimation
  - Insufficient understanding of code complexity
  - Insufficient provisioning for verification and obstacles
  - Refactoring often overruns in both time and budget
- Factors that can help
  - Knowing the scope and sticking to it
    - If there is change in scope estimate again
  - Plan for all stages of the process with contingency factors built-in
  - Make provision for developing tests and other forms of verification
    - Can be nearly as much or more work than the code change
    - Insufficient verification incurs technical debt

Cost estimation

When development and production co-exist

- Potential for branch divergence
- Policies for code modification
  - Estimate the cost of synchronization
  - Plan synchronization schedule and account for overheads
- Anticipate production disruption
  - From code freeze due to merges
  - Account for resources for quick resolution of merge issues

- In the heat example
  - No more than a few hours of developer time
  - No disruption
  - No need for a buy-in

This is where buy-in from the stake-holders is critical

How do we determine what other tests are needed?

Code coverage tools

- Expose parts of the code that aren't being tested
  - gcov - standard utility with the GNU compiler collection suite (we will use it in the next few slides)
  - Compile/link with –coverage & turn off optimization
  - counts the number of times each statement is executed
- gcov also works for C and Fortran
  - Other tools exist for other languages
  - JCov for Java
  - Coverage.py for python
  - Devel::Cover for perl
  - profile for MATLAB

Interoperability coverage may need something like the matrix in testing module

- Lcov
  - a graphical front-end for gcov
  - available at http://ltp.sourceforge.net/coverage/lcov.php
  - Codecov.io in CI module
- Hosted servers (e.g. coveralls, codecov)
- graphical visualization of results
- push results to server through continuous integration server

Exercise: Refactoring the Running Example

- Convert heatAll.C to the cleaner version with reusable code.
  - Though a solution exists and has been given to you your solution need not be identical
  - Think about how you want your final product to be and then go through the exercise of refactoring

- Here I am taking the clean solution that Mark wrote and generalizing the update_solution interface
  - Motivation: Do not want to change heat.C for adding another method
  - For this exercise we will use “ftcs” and “upwind15” as alternative options

Preparing for Refactoring – check coverage

- In your working copy add -coverage as shown below
- Run ./heat runame="ftcs_results"
- Run gcov heat.C
- Examine heat.C.gcov

```
HDR = Double.H
SRC = heat.C utils.C args.C exact.C ftcs.C upwind15.C crankn.C
OBJ = $(SRC:.C=.o)
GCOV = $(SRC:.C=.C.gcov) $(SRC:.C=.gcda) $(SRC:.C=.gcno) $(HDR:.
H=.H.gcov)
EXE = heat

# Implicit rule for object files
%.o : %.C
        $(CXX) -c -coverage $(CXXFLAGS) $(CPPFLAGS) $< -o $@

# Linking the final heat app
heat: $(OBJ)
        $(CXX) -coverage -o heat $(OBJ) $(LDFLAGS) -lm
```

Preparing for Refactoring – check coverage

- In your working copy add -coverage as shown below
- Run ./heat runame="ftcs_results"
- Run gcov heat.C
- Examine heat.C.gcov

```
HDR = Double.H
SRC = heat.C utils.C args.C exact.C ftcs.C upwind15.C crankn.C
OBJ = $(SRC:.C=.o)
GCOV = $(SRC:.C=.C.gcov) $(SRC:.C=.gcda) $(SRC:.C=.gcno) $(HDR:.
H=.H.gcov)
EXE = heat

# Implicit rule for object files
%.o : %.C
        $(CXX) -c -coverage $(CXXFLAGS) $(CPPFLAGS) $< -o $@

# Linking the final heat app
heat: $(OBJ)
        $(CXX) -coverage -o heat $(OBJ) $(LDFLAGS) -lm
```

- A dash indicates non-executable line
- A number indicated the times the line was called
- ##### indicates line wasn't exercised

```
    -:  143:static bool
  500:  144:update_solution()
    -:  145:{
  500:  146:    if (!strcmp(alg, "ftcs"))
  500:  147:        return update_solution_ftcs(Nx, curr, last, alpha, dx, dt, bc0, bc1);
#####:  148:    else if (!strcmp(alg, "upwind15"))
#####:  149:        return update_solution_upwind15(Nx, curr, last, alpha, dx, dt, bc0, bc1);
#####:  150:    else if (!strcmp(alg, "crankn"))
#####:  151:        return update_solution_crankn(Nx, curr, last, cn_Amat, bc0, bc1);
#####:  152:    return false;
  500:  153:}
    -:  154:
    -:  155:static Double
  500:  156:update_output_files(int ti)
    -:  157:{
  500:  158:    Double change;
    -:  159:
  500:  160:    if (ti>0 && save)
    -:  161:    {
#####:  162:        compute_exact_solution(Nx, exact, dx, ic, alpha, ti*dt, bc0, bc1);
#####:  163:        if (savi && ti%savi==0)
#####:  164:            write_array(ti, Nx, dx, exact);
#####:  165:    }
```

Preparing for Refactoring – get baselines

- Call to upwind15 not exercised
- Run ./heat alg="upwind15" runame="upwind_results

```
    -:  143:static bool
  500:  144:update_solution()
    -:  145:{
  500:  146:    if (!strcmp(alg, "ftcs"))
#####:  147:        return update_solution_ftcs(Nx, curr, last, alpha, dx, dt, bc0, bc1);
  500:  148:    else if (!strcmp(alg, "upwind15"))
  500:  149:        return update_solution_upwind15(Nx, curr, last, alpha, dx, dt, bc0, bc1);
#####:  150:    else if (!strcmp(alg, "crankn"))
#####:  151:        return update_solution_crankn(Nx, curr, last, cn_Amat, bc0, bc1);
#####:  152:    return false;
  500:  153:}
    -:  154:
```

- We have baselines for ftcs and upwind

```
[ahilya:clean dubey$ ls ftcs_results/
clargs.out                      ftcs_results_soln_00000.curve   ftcs_results_soln_final.curve
[ahilya:clean dubey$ ls upwind_results/
clargs.out                      upwind_results_soln_00000.curve upwind_results_soln_final.curve
ahilya:clean dubey$
```

Refactoring – The starting code

```
extern bool
update_solution_ftcs(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double bc_0, Double bc_1);

extern bool
update_solution_upwind15(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double bc_0, Double bc_1);

extern bool
update_solution_crankn(int n,
    Double *curr, Double const *last,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

```
if (!strncmp(alg, "crankn", 6))
    initialize_crankn(Nx, alpha, dx, dt, &cn_Amat);
```

- Interfaces are not identical
- crankn has an extra argument
- It also has an extra step in initialization

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

- Modify the makefile

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

- Modify the makefile

```
HDR = Double.H
SRC1 = heat.C utils.C args.C exact.C ftcs.C
SRC2 = heat.C utils.C args.C exact.C upwind15.C
SRC3 = heat.C utils.C args.C exact.C crankn.C
OBJ1 = $(SRC1:.C=.o)
OBJ2 = $(SRC2:.C=.o)
OBJ3 = $(SRC3:.C=.o)

EXE1 = heat1
EXE2 = heat2
EXE3 = heat3
```

Refactoring

- Generalize the interface

```
extern bool
update_solution(int n,
    Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1);
```

- Modify the makefile
- Add null implementations of initialize_crank in ftcs and upwind15

```
HDR = Double.H
SRC1 = heat.C utils.C args.C exact.C ftcs.C
SRC2 = heat.C utils.C args.C exact.C upwind15.C
SRC3 = heat.C utils.C args.C exact.C crankn.C
OBJ1 = $(SRC1:.C=.o)
OBJ2 = $(SRC2:.C=.o)
OBJ3 = $(SRC3:.C=.o)

EXE1 = heat1
EXE2 = heat2
EXE3 = heat3
```

Refactoring

```
void
initialize_crankn(int n,
    Double alpha, Double dx, Double dt,
    Double **_cn_Amat)
{
}

bool
update_solution(int n, Double *curr, Double const *last,
    Double alpha, Double dx, Double dt,
    Double const *cn_Amat,
    Double bc_0, Double bc_1)
{
    Double const f2 = 1.0/24;
    Double const f1 = 1.0/6;
    Double const f0 = 1.0/4;
    Double const k = alpha * alpha * dt / (dx * dx);
    Double const k2 = k*k;
```

- make heat1
- Run ./heat runame="ftcs_results"
- Make heat2
- Run ./heat runame="upwind_results"
- Verify against baseline

Graphical View of Gcov Output and Tutorials for Code Coverage

Overall Analysis

SOURCE FILES ON BUILD 45

LIST 2 CHANGED 0 SOURCE CHANGED 0 COVERAGE CHANGED 0

COVERAGE	Δ	FILE	LINES	RELEVANT	COVERED
74.39		src/functions/linear_fcn_class.f90	301	82	61
100.0		src/general/modulo_mod.f90	52	3	3

Detailed Analysis

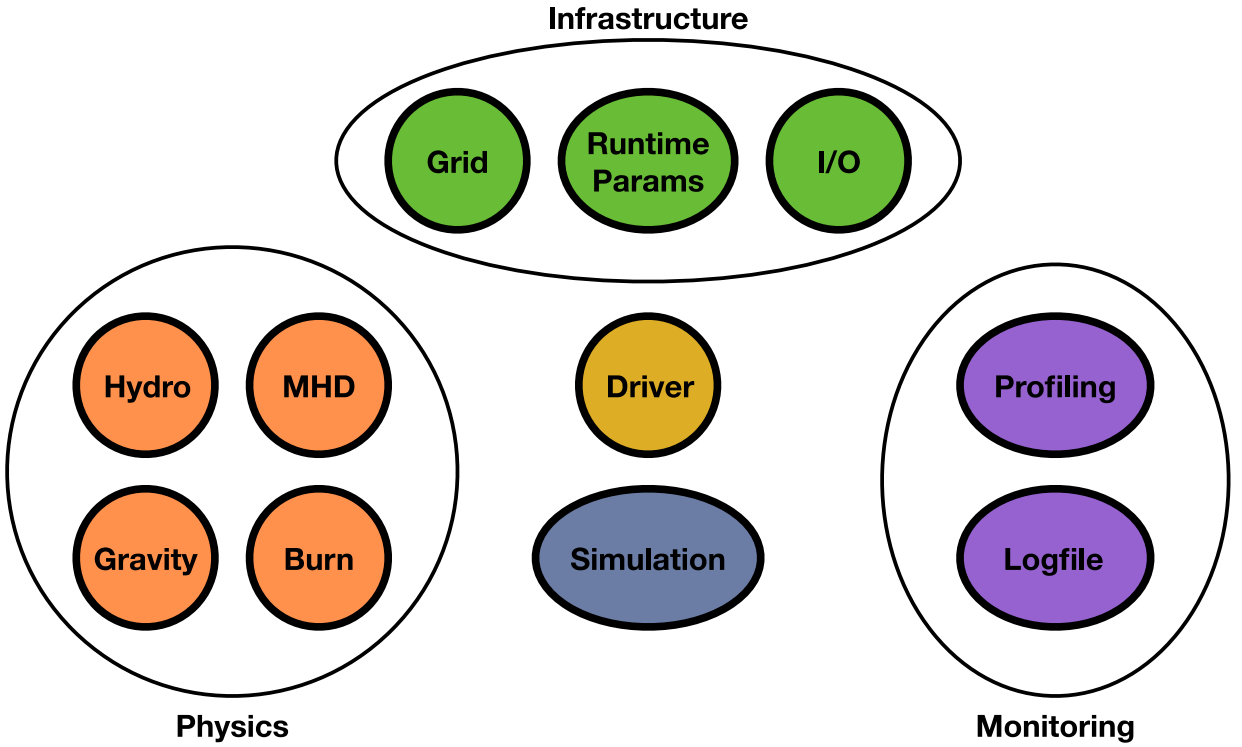
```
265         ! Error distribution same for all x values
266         delta = S*Sxx - Sx*Sx
267         if (delta == 0.0_wp) then
268             ERRORMSG("Cannot do linear least-sqrs.  Divide by zero.")
269             stop
270         end if
271         delta_inv = 1.0_wp / delta
```

Online tutorial - https://github.com/amklinv/morpheus

Other example - https://github.com/jrdoneal/infrastructure

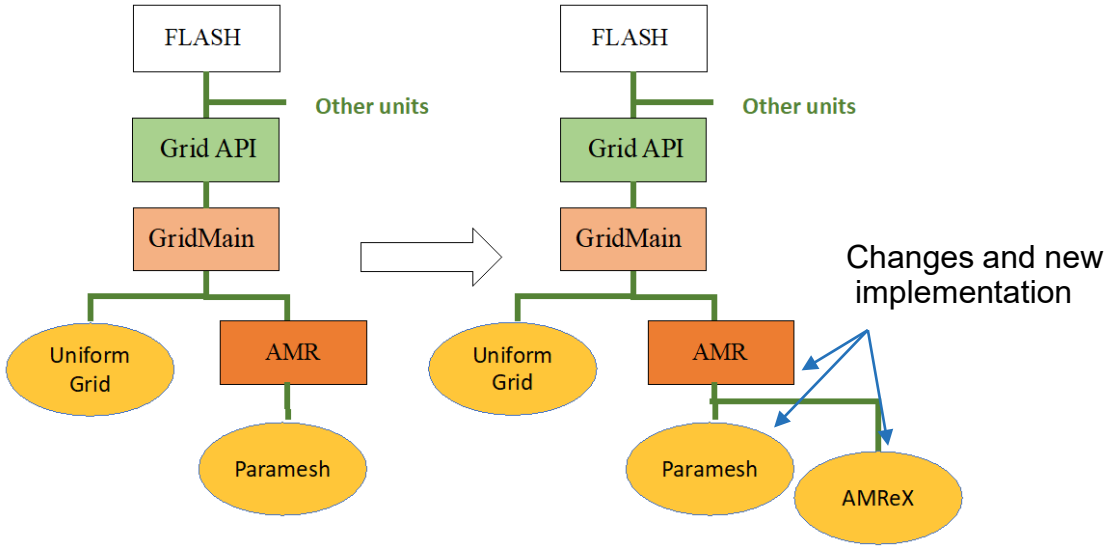
More Realistic Example From FLASH

- Grid
  - Manages data
  - Domain discretization
- Physics
  - `Several solvers`
- Driver
  - Time-stepping
  - Orchestrates interactions



More Realistic Example From FLASH

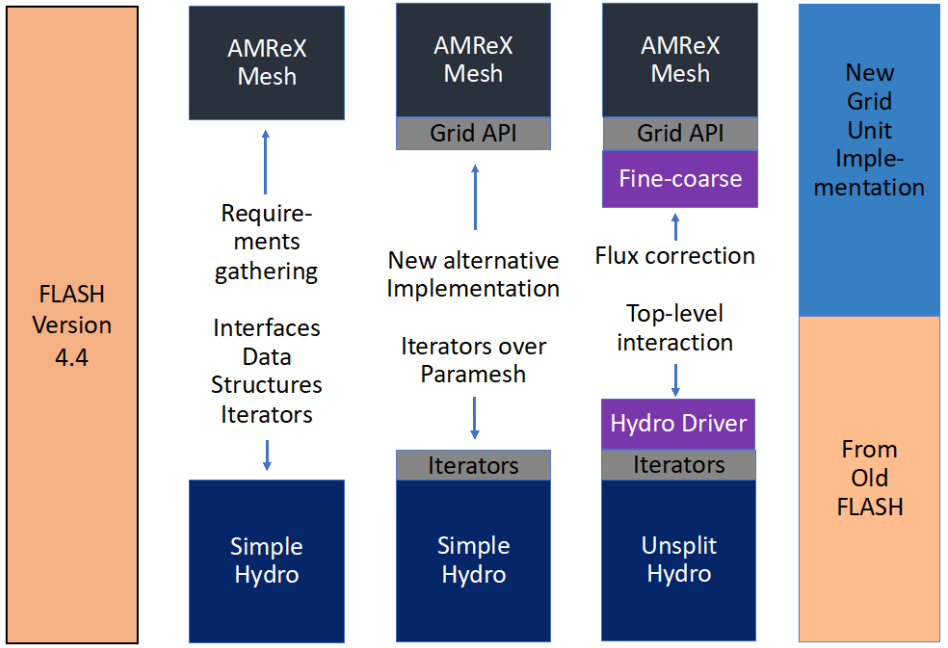
Goal: Replace Paramesh with AMReX



Considerations

- Cost estimation
  - Expected developer time
  - Extent of disruption in production schedules
- Get a buy-in from the stakeholders
  - That includes the users
  - For both development time and disruption

- In FLASH
  - Initial estimate at 6-12 months
  - Took close to 12 months



IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

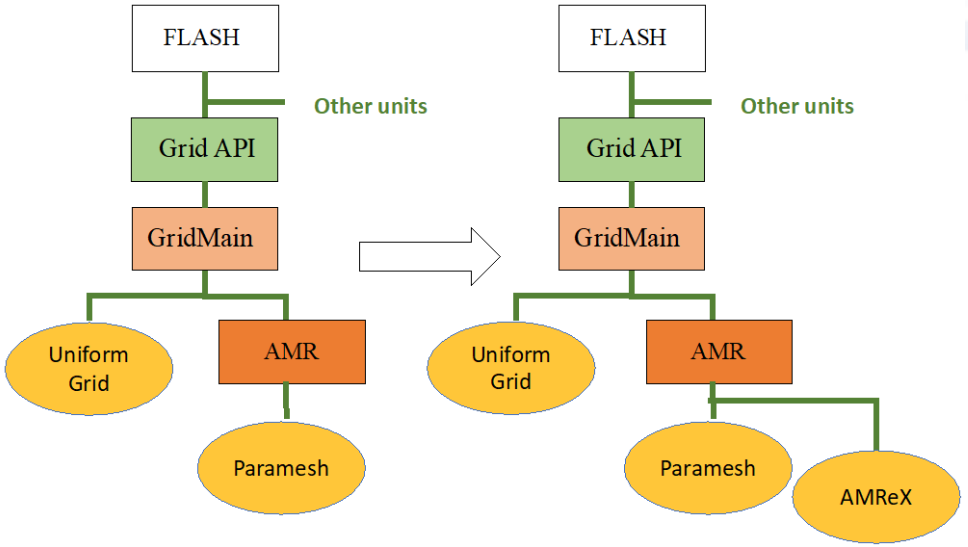
FLASH5

Refactoring for Next Generation Hardware

AMReX - Lawrence Berkeley National Lab
- Designed for exascale
- Node-level heterogeneity
- Smart iterators hide parallelization

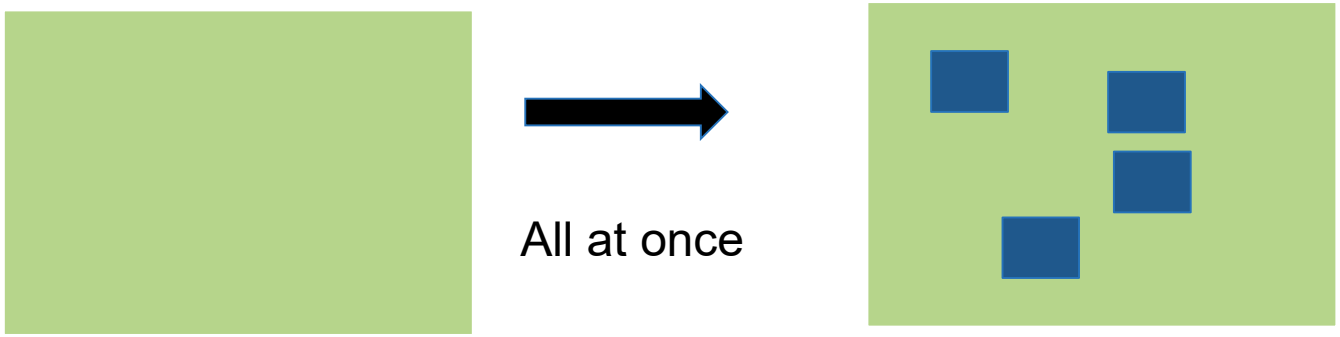
Goal: Replace Paramesh with AMReX

Plan: Getting there from here
- On ramping
- Design
- Intermediate steps
- Realizing the goal

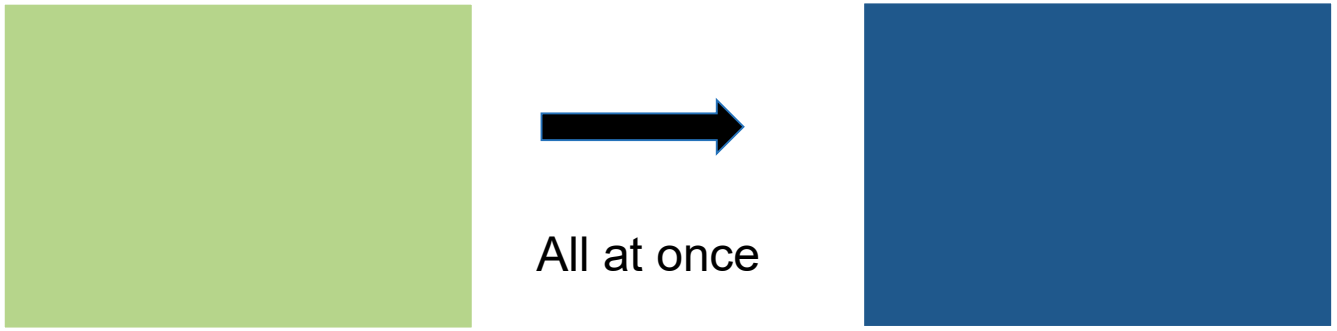


Map from Here to There: On ramp plan

Proportionate to the scope



Scattered independent changes - May be OK



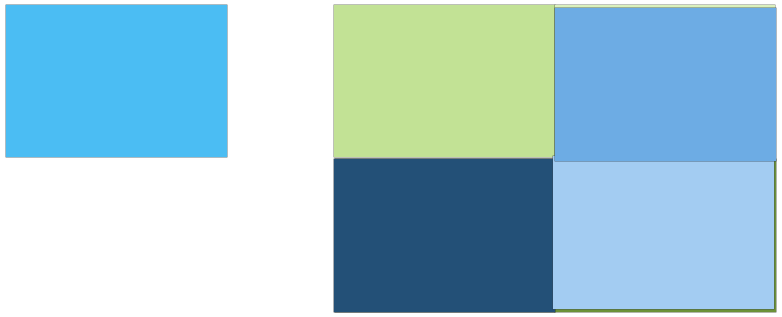
Invasive large-scale change in the code - Bad idea

IDEAS productivity

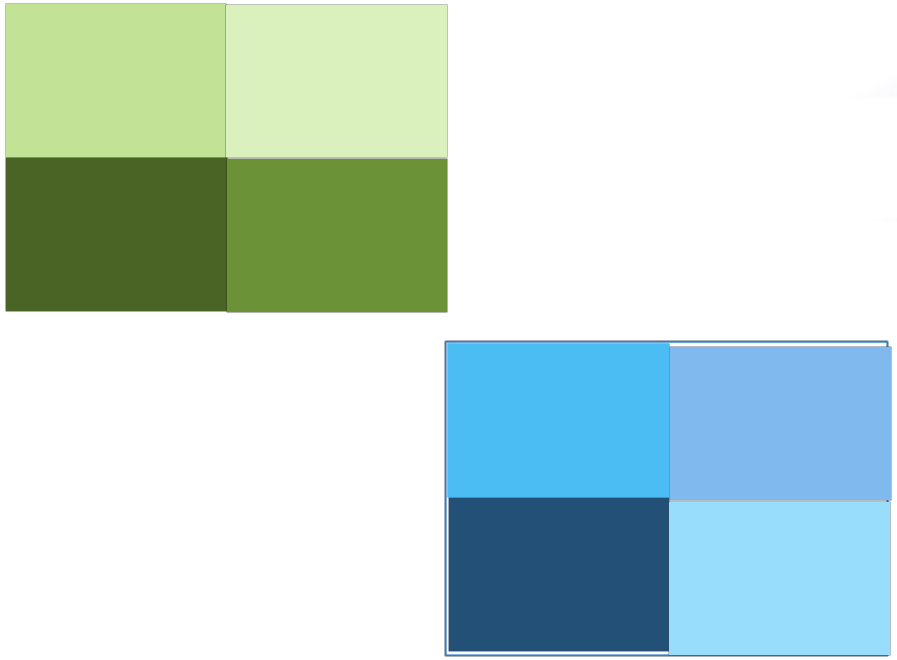
ECP EXASCALE COMPUTING PROJECT

On ramp plan

So how should it be done



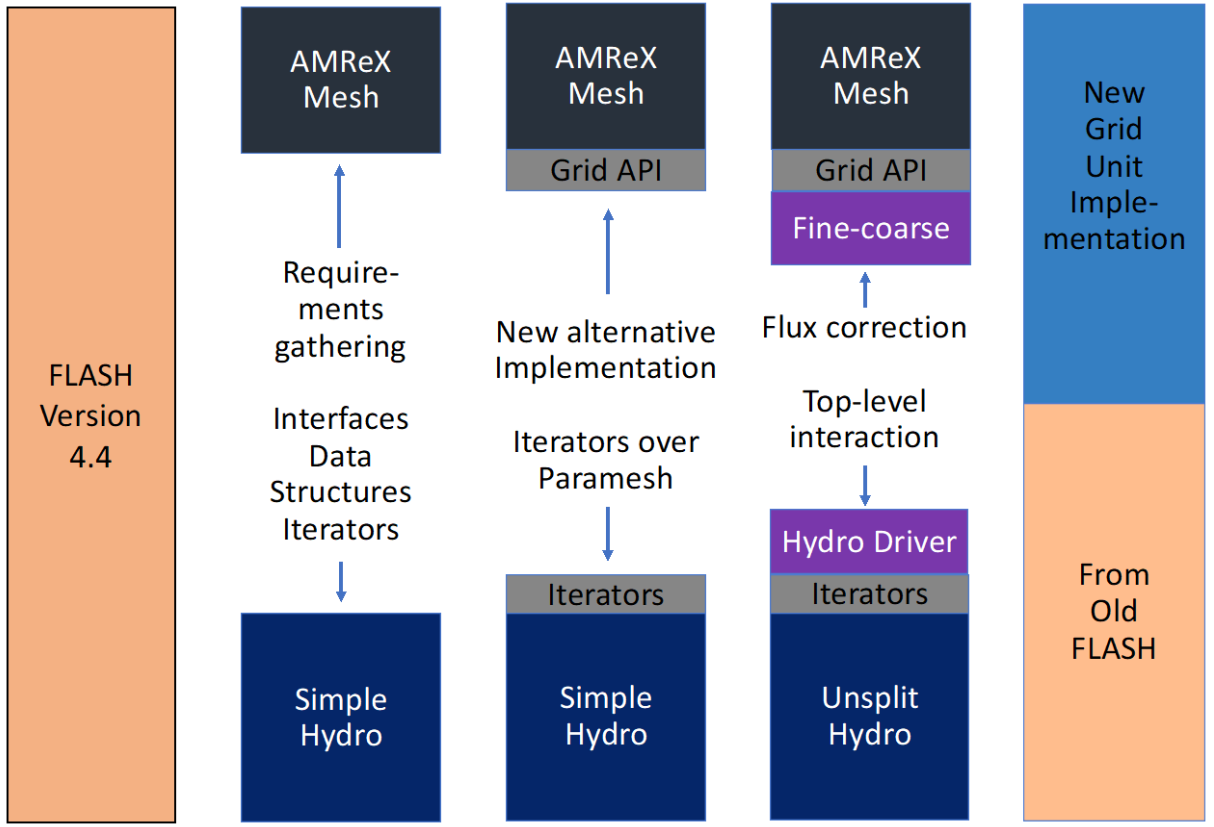
- Incrementally if at all possible
- Small components, verified individually
- Migrated back, integration tests done



- Alternatively migrate them into new infrastructure

Map From Here to There

- Paramesh & AMReX coexist
- Adapt interfaces to suit AMReX
- Refactor Paramesh implementation
- Compare AMReX implementation against Paramesh implementation



Refactoring plan

Design

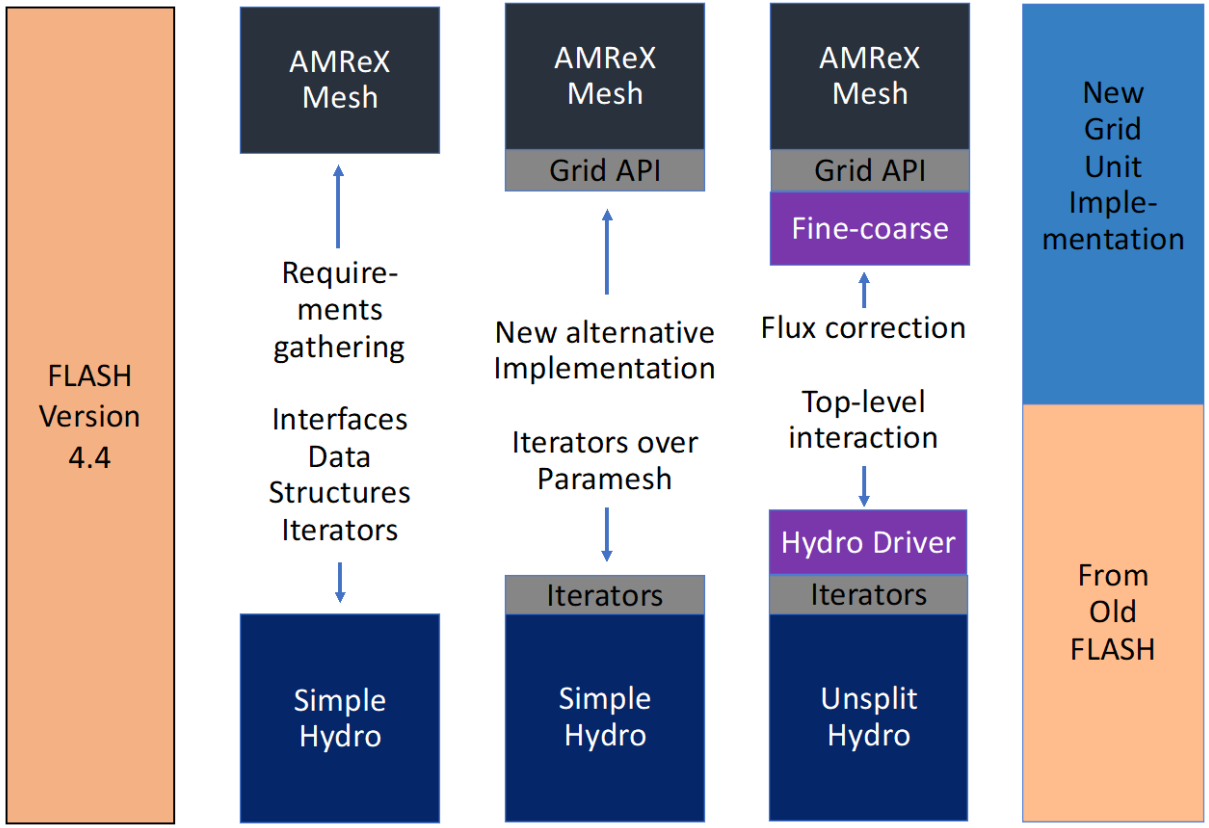
- Degree & scope of change
- Formulate initial requirements

Prototyping

- Explore & test design decisions
- Update requirements

Implementation

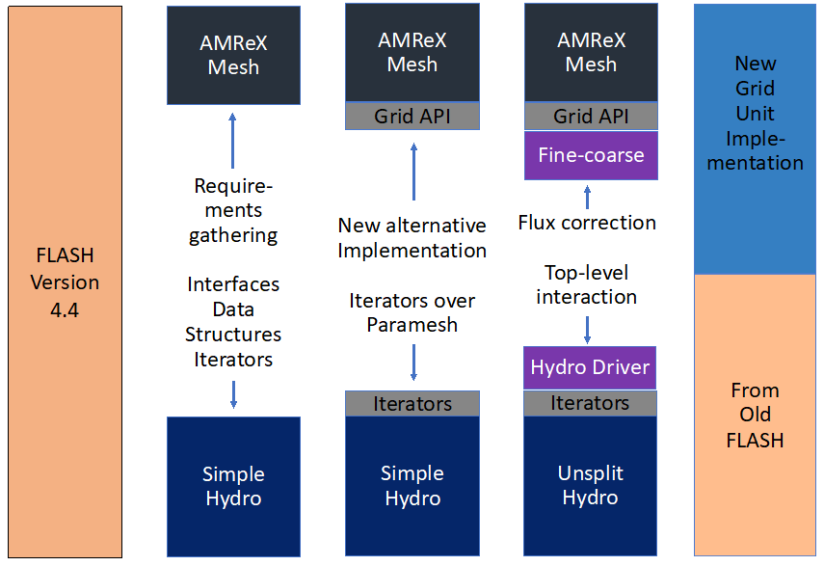
- Recover from prototyping
- Expand & implement design decisions



Phase 1 - design

Sit, think, hypothesize, & argue

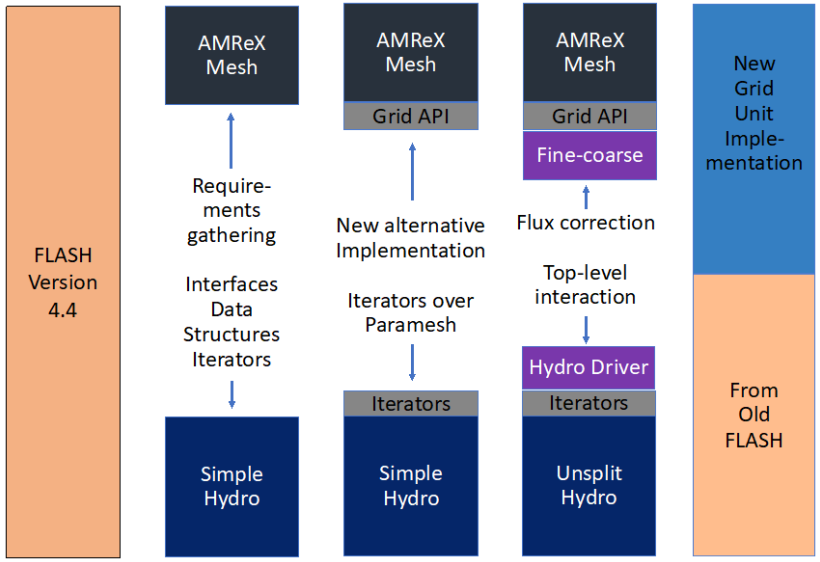
- Derive and understand principal definitions & abstractions
- Collect & understand Paramesh/AMReX constraints
  - Generally useful design due to two sets of constraints?
- Collect & understand physics unit requirements on Grid unit
- Design fundamental data structures & update interface



Phase 2 - prototyping

Quick, dirty, & light

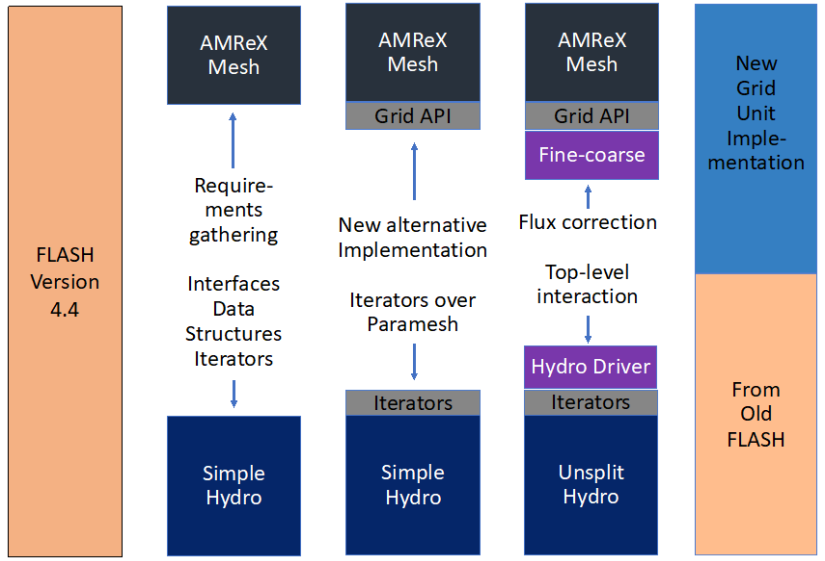
- Implement new data structures
  - Evolve design/implementation by iterating between Param
- Explore Grid/physics unit interface
  - `simpleUnsplit` Hydro unit
  - A simplified implementation
    - No need to be physically correct
    - Exercise the grid interface identically to the real solver
- Discover use patterns of data structures and Grid unit interface
- Adjust requirements & interfaces



Phase 3 - implementation

Toward quantifiable success & Continuous Integration

- Derive & implement lessons learned
  - Clean code & inline documentation
- Update `Unsplit` Hydro
- Hybrid FLASH
  - AMReX manages data
  - Paramesh drives AMR
- Fully-functioning simulation with AMReX
- Prune old code



IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Important Takeaways

Procedures and policies

- Developers should know what the end code should be
  - They will do the code implementation
  - You may need to develop some possibly throwaway code
    - Often that ends up being useful in unexpected ways

Process and policies are important

- Managing branch divergence
- Any code pruning
- Schedule of testing
- Schedule of integration and release
  - Release may be external or just to the internal users

Other resources

- Software testing levels and definitions:
  - http://www.tutorialspoint.com/software_testing/software_testing_levels.htm
- Working Effectively with Legacy Code, Michael Feathers.
  - The legacy software change algorithm described in this book is very straight-forward and powerful for anyone working on a code that has insufficient testing.
- Code Complete, Steve McConnell. Includes testing advice.
- Software Carpentry: http://katyhuff.github.io/python-testing/
- Tutorial from Udacity: https://www.udacity.com/course/software-testing--cs258
- Papers on testing:
  - http://www.sciencedirect.com/science/article/pii/S0950584914001232
  - https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH
- Resources for Trilinos testing:
  - Trilinos testing policy: https://github.com/trilinos/Trilinos/wiki/Trilinos-Testing-Policy
  - Trilinos test harness: https://github.com/trilinos/Trilinos/wiki/Policies--%7C-Testing

TO HAVE GOOD OUTCOME FROM REFACTORING

1. KNOW WHY
2. KNOW HOW MUCH
3. KNOW THE COST
4. PLAN
5. HAVE STRONG TESTING AND VERIFICATION
6. GET BUY-IN FROM STAKEHOLDERS