Argonne NATIONAL LABORATORY

SOFTWARE REFACTORING

ANSHU DUBEY
Mathematics and Computer Science Division
Argonne National Laboratory

August 8, 2016
ATPESC
St. Charles IL

ABOUT THIS PRESENTATION

- What this lecture is ---
  - Methodology for planning the refactoring process
    - Considerations before and during refactoring
    - Developing a workable process and schedule
    - Possible pitfalls and workarounds
  - Examples from codes that underwent refactoring
    - And their lessons learned

- What this lecture is not ---
  - Instructions on detailed process of refactoring
    - It is a difficult process
    - Each project has its own quirks and challenges
    - No one methodology will apply everywhere
  - Tutorial on tools for refactoring
    - There really aren't that many

BEFORE STARTING

CONSIDERATIONS

- Know why you are refactoring
  - Is it necessary
  - Where should the code be after refactoring
- Know the scope of refactoring
  - How deep a change
  - How much code will be affected
- Estimate the cost
  - 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

REASONS FOR REFACTORING

The big one these days is the change in platforms

- Once before
  - Vector to risc processors (cpu)
  - Flat memory model to hierarchical memory model
- To heterogeneous
  - Few CPU's sufficient memory per cpu
  - Several co-existing memory models
- The driving reason for these transitions is performance
  - Performance may drive refactoring even without change in platforms

REASONS FOR REFACTORING

There can be other reasons

- Transition of code from research prototype to production

- Imposing architecture and maintainability on an old code
  - Significant change in the code base
    - Change in model or discretization
    - Changes in numerical algorithms
  - Significant change in intended use for the code
    - From a small team to a large team
    - Releasing to wider user base

- Enabling extensibility or configurability
  - Partial common functionality among different usage modes
  - Model refinement
  - Incorporating new insights

SCOPE OF REFACTORING

Know where you want the end product to be

- For performance
  - Know the target improvement
    - Very easy to go down the rabbit hole of squeezing the last little bit
    - Almost never worth the effort for obtaining scientific results

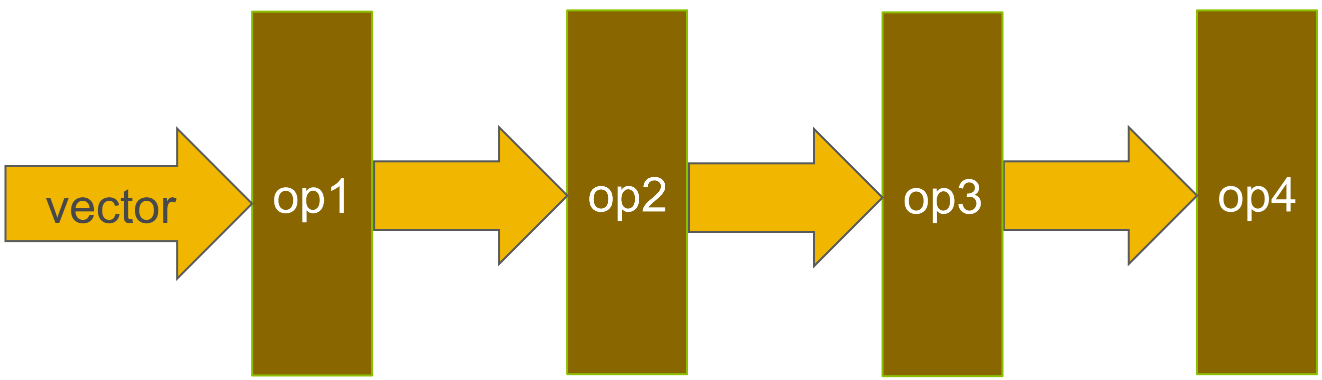
- For maintainability
  - Know the boundaries for imposing structure
    - Rewriting the entire code is generally avoidable
    - Kernels for implementing formulae can be left alone ?
    - In general it possible to stop at higher levels than that

- For extensibility
  - Similar to maintainability
  - Greater emphasis on interfaces and encapsulation

REASONS FOR REFACTORING

The big one these days is change in platforms

Transition from vector to risc machines



For vector processors

- Data structures needed to be long vectors
  - Longer => better
- Spatial or temporal locality had no importance
  - Memory access was flat
    - Interleaving banks for better performance

REASONS FOR REFACTORING

The big one these days is change in platforms

Transition from vector to risc machines

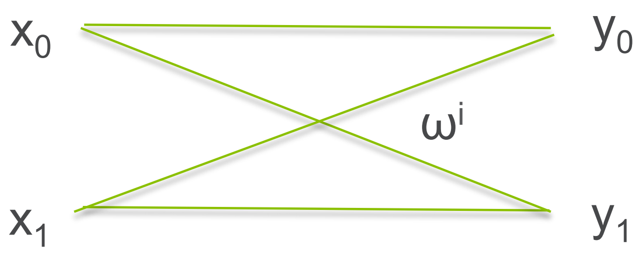
small chunk that could fit
In the cache

For risc processors

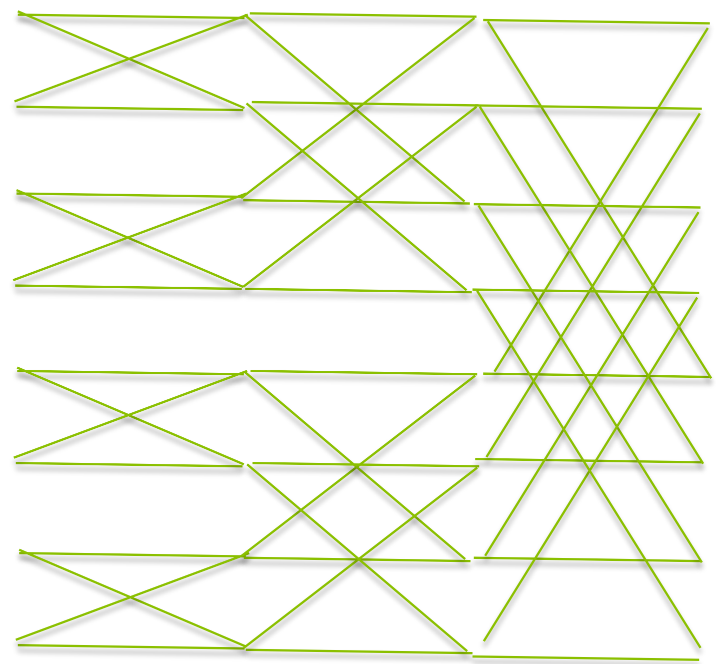
- Memory has hierarchy
  - Closer and smaller => faster access
  - Small working sets that can persist in the closest memory preferable
  - Makes spatial and temporal locality important
- Data structures that enable formation of small working sets on which multiple operations can be performed are better

HOW WOULD THE CODE CHANGE ?

Example of FFT calculation



$$y_0 = x_0 + \omega^i * x_1$$
$$y_1 = x_0 - \omega^i * x_1$$



VECTOR OPERATIONS

ω^0
.
.
$\omega^{n/2-1}$

$T_0=\omega^0 * x_1$
$T_1-\omega^0 * x_1$
.
.
.
.
$T_{n-2}=\omega^0 * x_{n-1}$
$T_{n-1}=-\omega^0 * x_{n-1}$

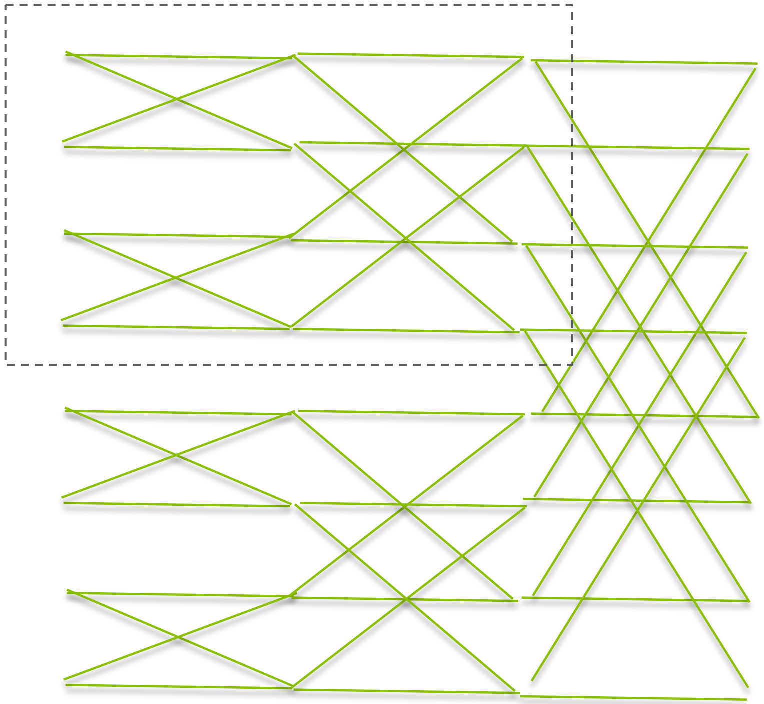
$T_0=x_0+T_0$
.
.
.
.
.
.
$T_{n-1}=x_{n-1}+T_{n-1}$

after some
permutations
....
and
computations

$T_0=\omega^0 * x_{n/2}$
$T_1=\omega^1 * x_{n/2+1}$
.
.
$T_{n/2-1}=\omega^{n/2-1} * x_{n-1}$
$T_{n/2}=-\omega^0 * x_{n/2}$
-
-
$T_{n-1}=-\omega^{n/2-1} * x_{n-1}$

$T_0=\omega^0 * x_1$
$T_1-\omega^0 * x_1$
.
.
.
.
$T_{n-2}=\omega^0 * x_{n-1}$
$T_{n-1}=-\omega^0 * x_{n-1}$

RISC CALCULATION



- Order of operations changes
- Loops need rearranging
- Extra nesting in loops may be required

Assume cache accommodates working set for k butterflies at a time

- Blocking of input vector
  - first $\log_2 k+1$ stages computed in one block
  - then shuffle so that next $\log_2 k+1$ stages can be computed

$x_0,x_1 \ldots\ldots\ldots x_{14},x_{15}$

↓

$x_0,x_4,x_8,x_{12},x_1,x_5 \ldots\ldots\ldots x_{11},x_{15}$

  - Repeat until done

Note that vector algorithm would still have worked but would have been slow

PLANNING AND IMPLEMENTATION

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

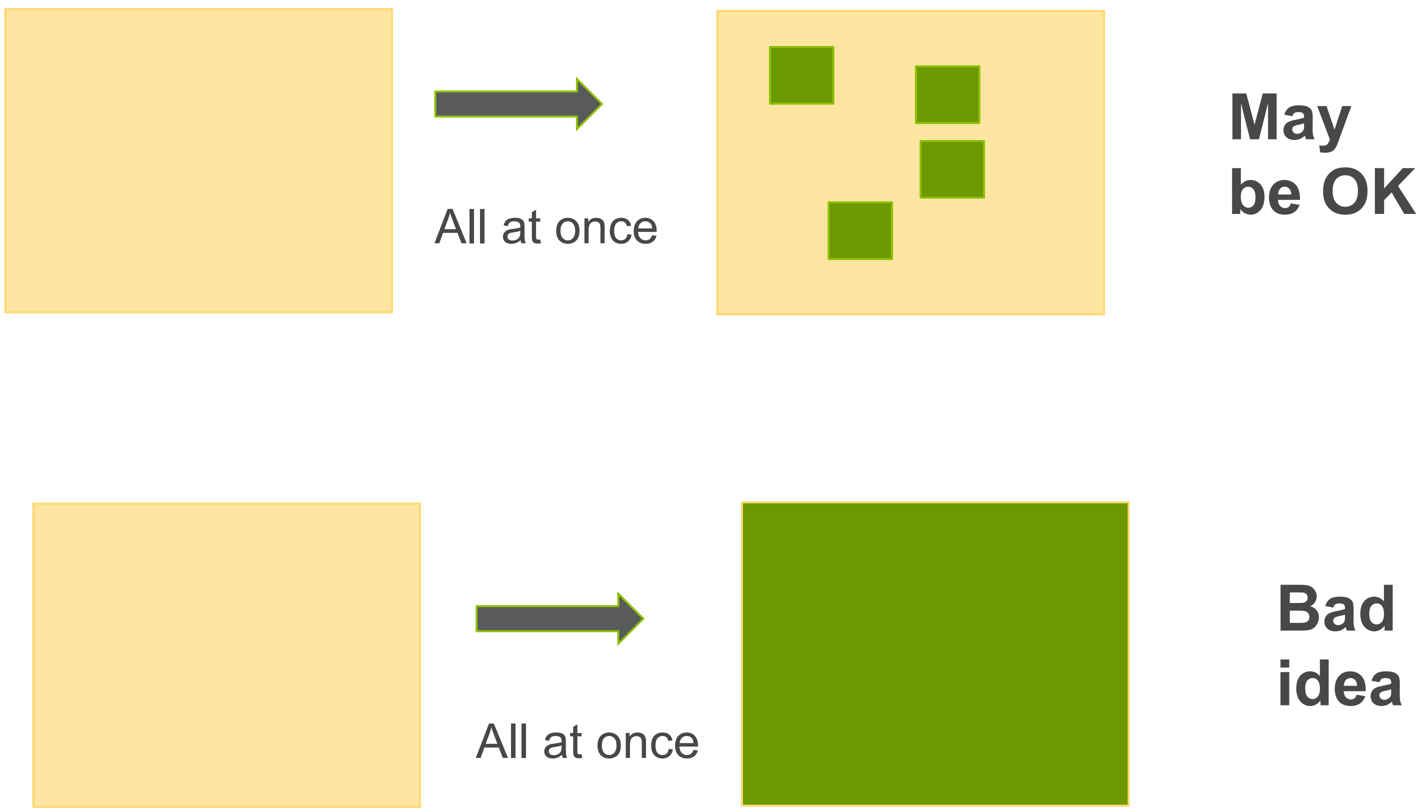
When development and production co-exist

- 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

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

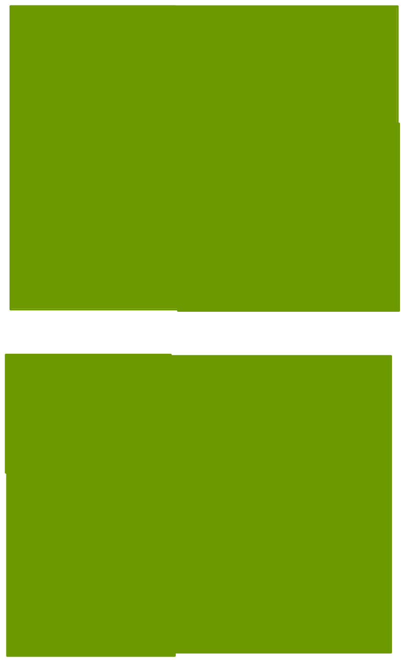
ON RAMP PLAN

Proportionate to the scope



ON RAMP PLAN

So how should it be done



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

- Alternatively migrate them into new infrastructure

VERIFICATION

Critical component of refactoring

- Understand the verification needs during transition
- Map from here to there
- Know your error bounds
  - Bitwise reproduction of results unlikely after transition
- Check for coverage provided by existing tests
- Develop new tests where there are gaps
- Make sure tests exist at different granularities
  - There should definitely be demanding integration and system level tests

IMPLEMENTATION

Procedures and policies

- Developers (hopefully) know what the end code should be
  – They will do the code implementation

Process and policies are important

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

EXPERIENCE – FLASH VERSIONS 1-4

Argonne NATIONAL LABORATORY

TRANSITION FROM VERSION 1-2

Version 1

- The Good
  - Desire to use the same code for many different applications necessitated some thought to infrastructure and architecture
  - Concept of alternative implementations, with a script for some plug and play
  - Inheriting directory structure to emulate object oriented approach
- The Bad
  - F77 style of programming; Common blocks for data sharing
  - Inconsistent data structures, divergent coding practices and no coding standards
- And the ugly
  - Two camps with divergent views
    - The science centric view won out
    - Capabilities got added while the worst of f77 remained

VERSION 2 : DATA INVENTORIED

- Objective was to make the code modular and extensible
  - Inventory the data,
  - Eliminate common blocks, classify variables
  - Introduce automated testing
- Objectives partially met
- Centralized database was built
  - It met the data objectives
  - But got in the way of modularization
  - No data scoping, partial encapsulation
  - Database query overheads

VERSION 3 : THE CURRENT ARCHITECTURE

- Kept inheriting directory structure, configuration and customization mechanisms from earlier versions
- Defined naming conventions
  - Differentiate between namespace and organizational directories
  - Differentiate between API and non-API functions in a unit
  - Prefixes indicating the source and scope of data items
- Formalized the unit architecture
  - Defined API for each unit with null implementation at the top level
- Resolved data ownership and scope
- Resolved lateral dependencies for encapsulation
- Introduced subunits and built-in unit test framework

VERSION 4

Capability building exercise

- Did not need any change in the architecture
- Few infrastructure changes
  - Mesh replication was easily introduced for multigroup radiation
  - Laser drive
  - Interface with linear algebra libraries
- No or minimal changes to existing code

VERSION TRANSITIONS

1-2: objectives partially met

- The bias at the time – keep the scientists in control
- Keep the development and production branches synchronized
  - Enforce backward compatibility in the interfaces

Reasons for only partial success

- Too much synchronization between branches
  - Precluded needed deep changes
  - Hugely increased developer effort
  - High barrier to entry for a new developer
- Not enough buy-in from users
  - Did not get adopted for production in the center for more than two years
    - Development continued in FLASH1.6, and so had to be brought simultaneously into FLASH2 too

VERSION TRANSITIONS

From 2-3

- Build the framework in isolation from the production code base
  - Used the second model in the ramp-on slide
- Ramp on was planned, scope of change was determined ahead of time, scientists were on-board with the plan
- The ramp on plan
  - Infrastructure units first implemented with a homegrown Uniform Grid.
  - Unit tests for infrastructure built before any physics was brought over
  - Test-suite started on multiple platforms
  - Migrate mature solvers (few likely changes) and freeze them in version 2
  - Migrate the remaining solvers one application dependencies at a time
  - Scientists in the loop for verification and in prioritizing physics migration

There was no well defined transition from version 3 to 4 because it was mostly adding code

TO HAVE GOOD OUTCOME FROM REFACTORING

KNOW WHY

KNOW HOW MUCH

KNOW THE COST

PLAN

HAVE STRONG TESTING AND VERIFICATION

GET BUY-IN FROM STAKEHOLDERS