

Refactoring with a Case Study

Presented to
ATPESC 2018 Participants

Anshu Dubey

Computer Scientist, Mathematics and Computer Science Division

Q Center, St. Charles, IL (USA)

Date 08/08/2018



EXASCALE COMPUTING PROJECT



U.S. DEPARTMENT OF
ENERGY

Office of
Science



License, citation, and acknowledgments

License and Citation



- This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/) (CC BY 4.0).
- Requested citation: Anshu Dubey, Refactoring with a case study, tutorial, in Argonne Training Program on Extreme-Scale Computing (ATPESC) 2018. DOI: 10.6084/m9.figshare.6943076.

Acknowledgements

- 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 managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357

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 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

Definition

The general definition of refactoring

Refactoring usually applies to object oriented software where the internals of the implementations are “cleaned up” without changing the behavior.

In the context of this lecture

A broad interpretation where any part of the software may change while retaining or enhancing its basic capabilities.

The reason

In context of HPC scientific software the degree of change is motivated by many factors. It may include redesign at a higher level.



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

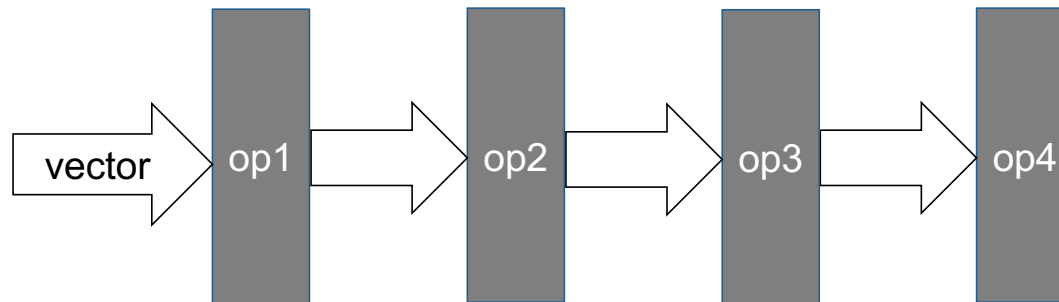
Know where you want the end product to be

- 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 extensibility
 - Similar to maintainability
 - Greater emphasis on interfaces and encapsulation
- 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 is possible to stop at higher levels than that

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

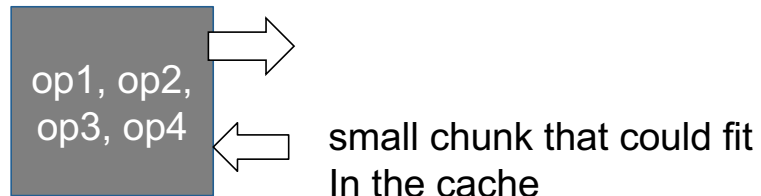
Reasons for refactoring

The big one these days is change in platforms

Transition from vector to risc machines

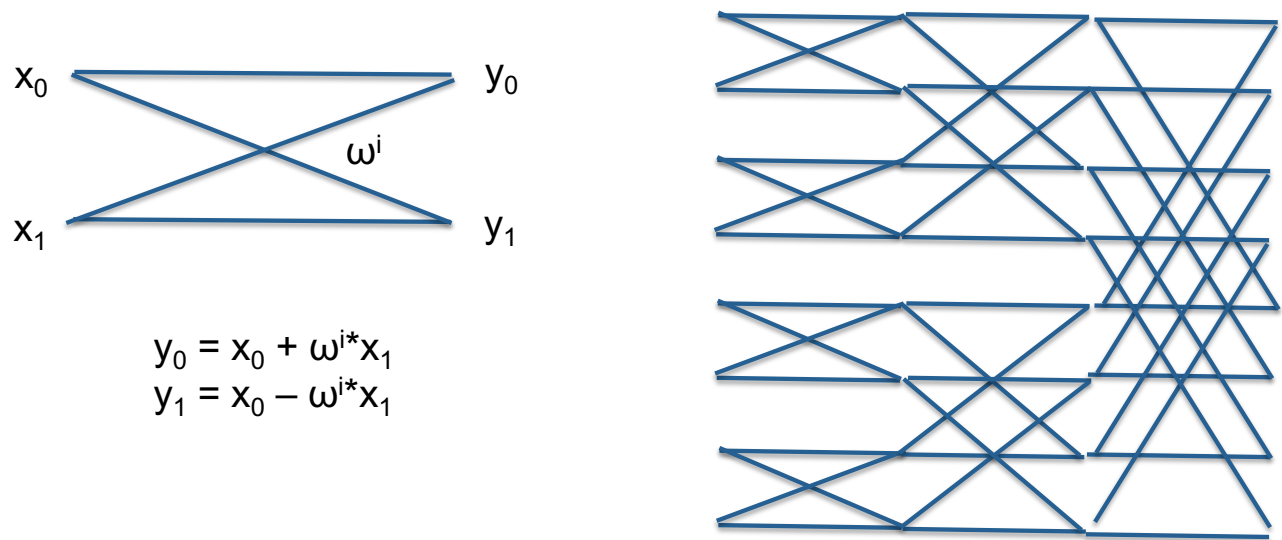
For risc processors

- 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

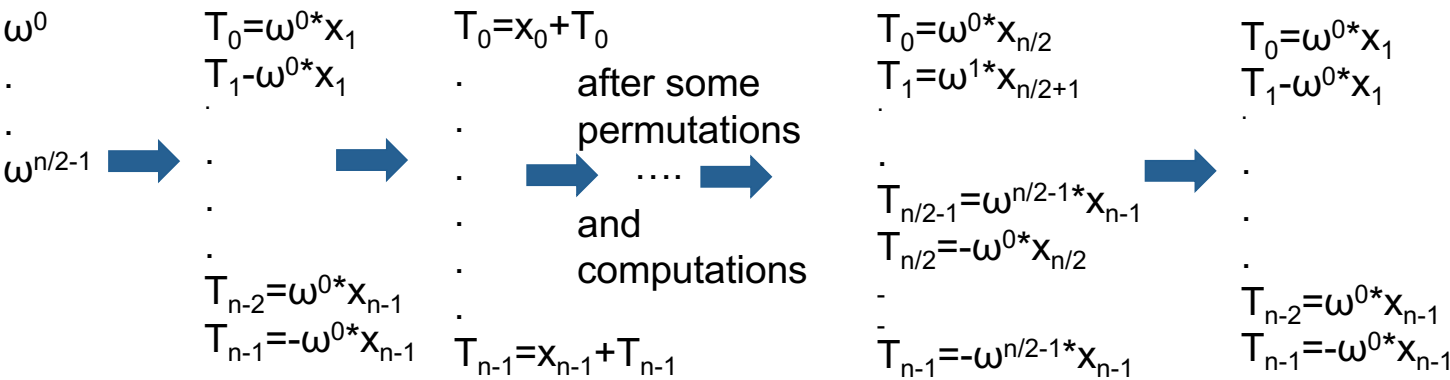


How would the code change ?

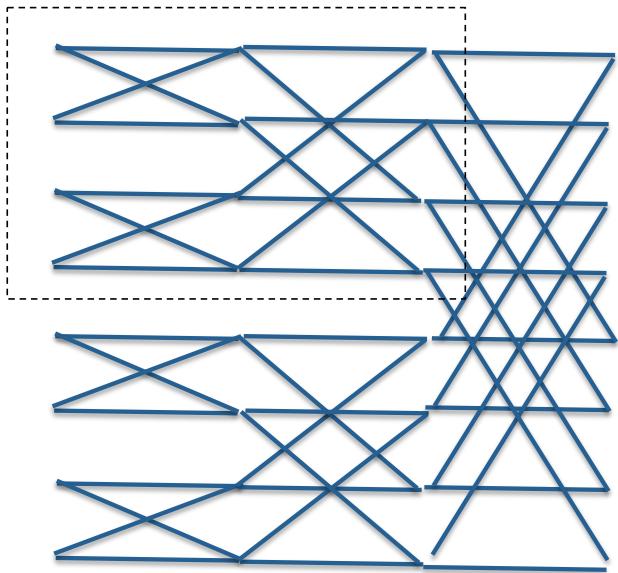
Example of FFT calculation



vector operations



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

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



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

- 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

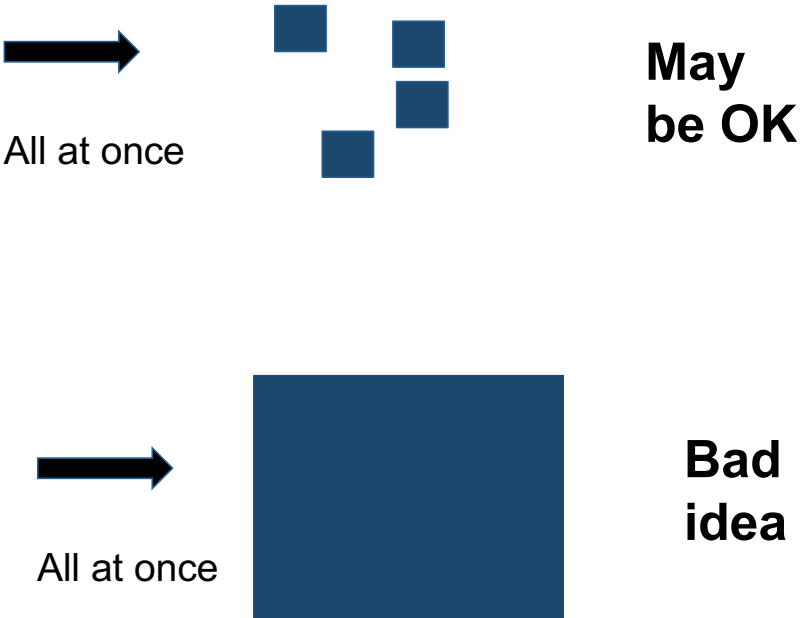
When development and production co-exist

- 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 1-5

Version 1

- Three independently developed codes smashed together
 - Desire to use the same code for many different applications necessitated some thought to infrastructure and architecture
- Challenges
 - F77 style of programming; Common blocks for data sharing
 - Inconsistent data structures, divergent coding practices and no coding standards
- Solution
 - A setup script and config files
 - Concept of alternative implementations, with a script for some plug and play
 - Inheriting directory structure to emulate object oriented approach
 - Wrapper layer with interfaces

Version 2

- Data inventory and interface formalization
 - Modularize the code and make it extensible
 - Elimination of common blocks
 - Formalization of interfaces
- 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
- Scope not fully determined
 - Enforced backward compatibility
 - 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 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 transition

- Build the framework in isolation
 - Used the second model in the ramp-on slide
- Ramp on was planned
 - scope of change was determined ahead of time
 - Determine data scoping and arbitration
 - Code mostly not altered at the kernel level
 - Base APIs for various units
 - scientists were on-board with the plan
 - Including the depth of changes

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

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

No explicit version transition methodology

Version 5

Ongoing

- Objective: prepare for platform and deeper heterogeneity
 - Expected changes in platforms
 - Hierarchical parallelism
 - Remove bulk synchronism
 - Different targets for execution
 - Needed in the code
 - Deeper encapsulation of physics kernels
 - Knowledge of grid
 - Constrained semantics
 - Enable code transformation and optimization

Version 5

- Approach
 - Determine level of modifications for each aspect of the change in code
 - Where possible keep modifications orthogonal between different aspects
 - Determine changes to setup script, config and API
 - Devise an approach to prototyping
 - Devise verification methodology
 - Add tests as needed
 - Devise an approach for moving from prototyping to production code

Version 5

- Implementation
 - Change looping over blocks to smart iterators
 - Metadata obtained through the iterator
 - The iterators can be looping over arbitrary sections of the domain
 - Metadata ensures that the physics kernels only see the domain they are meant to operate on
 - Add function calls in place of explicit statements where possible
 - The overhead of function call can be eliminated through code translation
- Iterators and function calls in kernels do not interfere with one another

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

Questions



EXASCALE COMPUTING PROJECT



U.S. DEPARTMENT OF
ENERGY

Office of
Science

