ExxonMobil

SOS26 Workshop, March 12, 2024

Outrunning the Red Queen

GettyImages-463495989

Vadim Dyadechko

Why do we HPC?

HPC exists to help us humans deal with complex phenomena around us

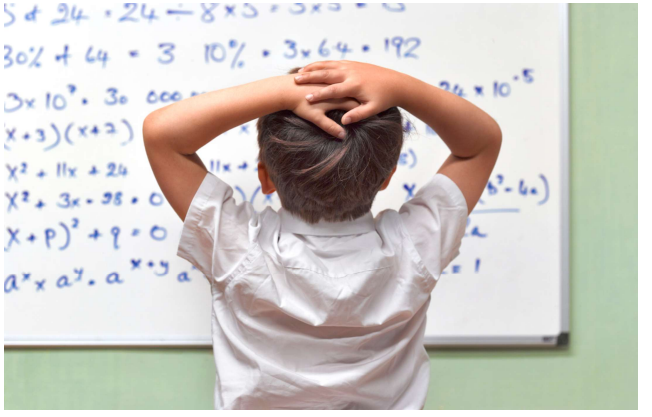
- Humans are not good at precise sciences:

  as species, we evolved to make fast

  ball-park-(in)accurate estimates

  "Every complex problem has an obvious

  yet wrong solution"



GettyImages-826680702

Pandora box of affordable compute:

- Complex phenomena, complex tools, complex input parameter space

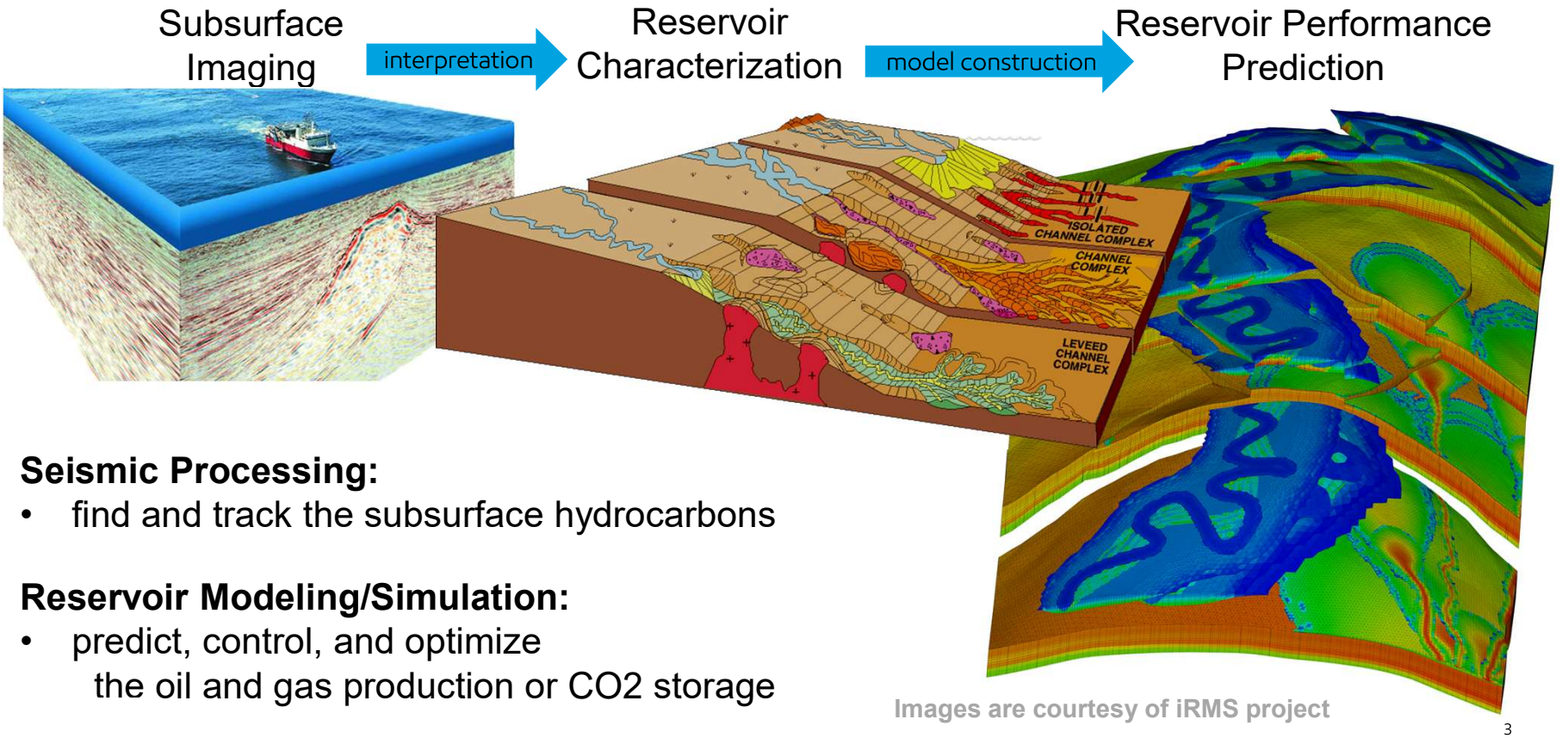
  "We're sitting on 4M pounds of fuel, 1 nuclear weapon and a thing that has 270K moving parts built by the lowest bidder" *(Armageddon)*

- Do we really have more predictive power today than yesterday?

  "There are million ways to generate garbage results,

  there is only a handful of ways to get them right"

HPC for Oil and Gas Upstream business



Seismic Processing:

- find and track the subsurface hydrocarbons

Reservoir Modeling/Simulation:

- predict, control, and optimize the oil and gas production or CO2 storage

Images are courtesy of iRMS project

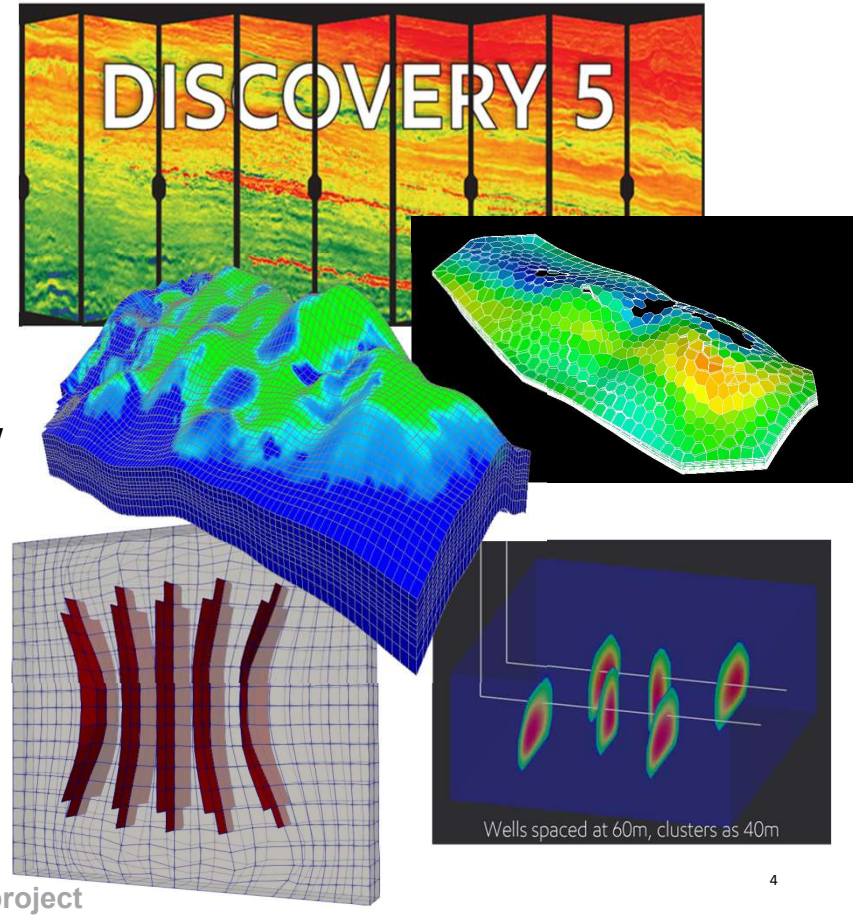
HPC for Oil and Gas Upstream business

Three whales (well, hogs):

- **Seismic imaging:**
  - acoustic/elastic wave propagation
  - FD discretization, explicit time marching
  - inverse problem: PDE-constrained optimization

- **Reservoir modeling and simulation:**
  - multi-phase multi-component pipe/porous media flow
  - FV discretization, implicit time marching
  - large-scale (non)linear solver

- **Geomechanic + fracking simulations:**
  - poro-elastic deformation + fracture propagation
  - FE/FV discretization, implicit time marching
  - large-scale (non)linear solver



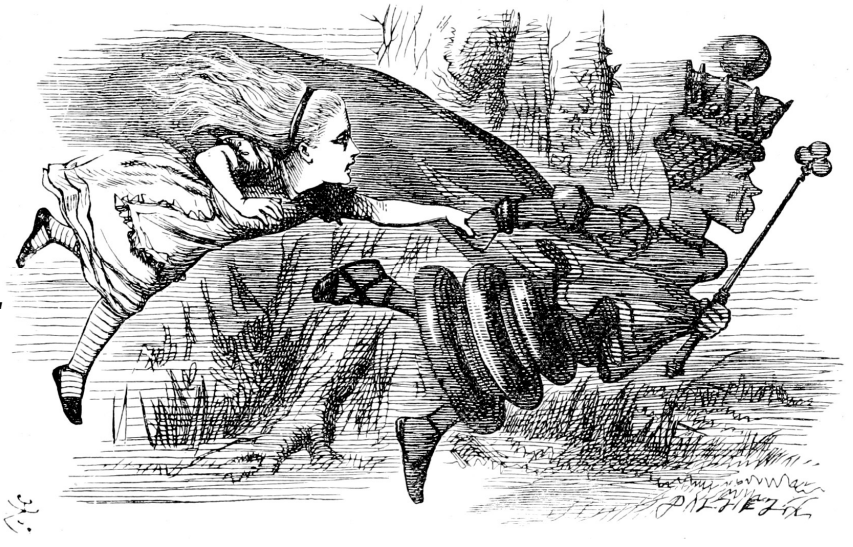
Images are courtesy of Seismic Processing, HPC Systems, and iRMS project

Red Queen's Races

"Well, in our country," said Alice,
"you'd generally get to somewhere else,
if you run very fast for a long time,
as we've been doing."

"A slow sort of country!" said the Queen.
"Here it takes all the running you can do,
to keep in the same place.
If you want to get somewhere else, you
must run at least twice as fast as that!"

(Lewis Carroll, Through the Looking-Glass, illustrated by John Tenniel)



GettyImages-463495989

The Loneliness of the Long Distance Runner

HPC bottlenecks (in ascending order of severity):

- Instruction pipeline
- Memory bandwidth
- Network bandwidth
- Storage throughput
- Power source, heat sink
- People who develop, deploy, and run s/w ?

GettyImages-172194820

We are so focused on physical bottlenecks
that we are missing the fact that human nature
slowly becomes a dominating productivity/risk factor:

we, humans, are
the least scalable and the least reliable
component of HPC (well, of any) technology.

In a long run, we are essentially in the race
against ourselves

Software reflects human mentality

While s/w is stored and run on computers,
it is developed only by human brains.
Software is illusive because it is a mental product.

"It's all in your head" means *"You are out of touch with reality"*

- Intangible objects look to us less constrained than the real ones
  (think of a teenager with a new credit card)
- Our ballpark estimates always ignore the "distribution tail"
  (we tend to estimate the total of a long shopping list as the sum of large items)

1e6 + 1e6 + 1 + 1/2 + 1/3 + 1/4 + 1/5 +... ≈ 2e6 ? Wrong: infinity!

How can we expect to create a perfect s/w product
if we constantly fall into the traps set up by our own brains?

Human factor: “Here be dragons”

Human brains are not wired to handle complex s/w systems;
we are not always aware of our physiological limitations,
and too proud to admit that we are not in full control.

We constantly expand our capabilities
by improving the code structure
and automating development process,
but our appetites grow way faster than our powers.

Humans tend to have *unrealistic expectations* about s/w,
are *over-optimistic* in our plans, *systematically* under-estimating

- the development / testing / integration / operational cost
- the complexity of code and stability of operational environment

Awareness of our own fallacies
is the first step towards sustainable s/w development.

Managing expectations: hardware vs software

"Why is it so much harder to manage software projects than hardware projects?"
(Thomas Watson Jr., IBM)

We don't talk about hardware sustainability because:

- hardware is tangible
- hardware is final (static)
- hardware is fit for purpose
- hardware is well documented
- hardware has well defined lifetime
- hardware has well defined operational environment:
  interfaces, protocols, tolerances, etc.
- hardware quality is taken very seriously (prohibitive cost of failure)

Hardware is quite rigid, we accept it and deal with it, no complains.

Because we know that the laws of physics assume no tolerances, the reality is simply unforgiving.

Managing expectations: hardware vs software

Software, on the other hand, is perceived / expected to

- be intangible
- be flexible and customizable
- accommodate more and more new features
- be fairly portable, work on a wide range of platforms
- be compatible with other pieces of software
- be of low maintenance
- eventually become bug-free (well, usable at least)
- be sprinkled with pixie dust?

Unfortunately, a lot of these expectations are over-inflated.

Managing expectations: hardware vs software

Here's a quick reality check:

- 3rd party compatibility requires fair amount of fitting and testing against components you do not control
- portability complicates the source code, build framework, multiplies the cost of testing
- most production s/w designs are surprisingly rigid
- complex codes can only be maintained by their authors
- every little feature increases overall complexity
- complexity breeds and shelters bugs
- complexity hogs time

Every wish has hidden cost, frequently substantial, sometime prohibitive.

But it does not stop us from requesting or promising "full service".

Such is the irresistible magic of word "soft".

Managing expectations: be realistic = be pessimistic?

To a large extent, the s/w sustainability problem is a problem of our unrealistic expectations.

A: "Oh, come on! You can't fit into this costume!"
B: "Is it Lycra?"
A: "Yes, but it's still governed by the laws of physics." (Just Shoot Me)

Since we don't know how far we can "stretch Lycra" without tearing it,
and we are known to have optimistic bias,
it is safer to wear a pessimistic hat (well, helmet) all the time.

We are much better at managing real objects, let's manage software as such:

- settle for a fit-for-purpose specification
- be conservative in expanding the scope
- take quality seriously

(non-essential) complexity is your enemy #1

- **Each system is only as robust as it is simple**
  - learn to be principled about requirements
    compromising code simplicity for incremental benefits
  - minimize the number of custom configuration files/parameters/options,
    introduce smart transparent defaults instead
  - **by all means avoid the trap of multi-dimensionality**
- **The key to success is self-imposed sharp focus:**
  - stick to one OS (Linux only)
  - stick to one compiler suite: GCC, LLVM, Intel, PGI -- your call
  - stick to mainstream hardware (multi-core CPUs)
  - stay away from multi-threading (MPI only)
  - stay away from high-maintenance 3rd-party dependencies (%42oost)
  - stay conservative in terms of tools/standards (make, c++11)
  - resist the temptation of writing a general-purpose library (3x cost)

Post-release support and refactoring: part of the package

Both code and operational environment are constantly evolving

- One cannot write a piece of code and leave it alone;
  it will eventually become obsolete/unusable
- S/w requires ever increasing support effort, which is not popular:
  - gets less visibility and recognition than adding new features
  - executed in a damage control mode rather than well planned manner
- S/w also requires timely re-factoring:
  - relieves the “internal stress”
  - allows to control complexity of the project

Post-release support and refactoring are unavoidable and consume fair amount of resources; plan and execute them accordingly.

Think outside the code

Coding is just one part of the larger picture;
there are many other aspects of s/w technology
that constantly toll the budget and beg to be improved.

There is a bigger prize is in optimizing
the entire s/w product pipeline / ecosystem.

There is a widespread misconception that s/w is ready for use
once the coding is completed,
and putting s/w to work is a trivial routine step
that somehow happens on its own.

Murphy's Law: "whatever can go wrong, will go wrong"

"Things will go wrong in any given situation, if you give them a chance"

- build is broken due to the system / library update
- production cases fail after the new release
- recent trunk updates break downstream integration tests
- tests passing on developer's machine but failing on the production server
- today's results are different from the ones obtained yesterday
- seemingly identical inputs A and B yield different results

These mishaps / mysteries / inefficiencies
are quite common;
and, **if left unchecked**, they start steadily
hog your time, drain resources,
and destroy your reputation.

GettyImages-538334833

"A horse! A horse! My kingdom for a horse!" (Richard III)

*"**For want of a nail** the shoe was lost.*
For want of a shoe the horse was lost.
For want of a horse the rider was lost.
For want of a rider the message was lost.
For want of a message the battle was lost.
*For want of a battle **the kingdom was lost.***
And all for the want of a horseshoe nail."

(13th century A.D.)

GettyImages-1132749725

The old nursery rhyme below a perfect example of how **simple, invisible, everyday problems can derail a critical mission**:

- 1991: a Patriot missile missed the target due to a round-off error
- 1996: sneaky type cast triggered self-destruction of Ariane 5 rocket
- 1999: NASA lost Mars Climate Orbiter due to the units disagreement

"Failure is Not an Option" (Apollo 13)

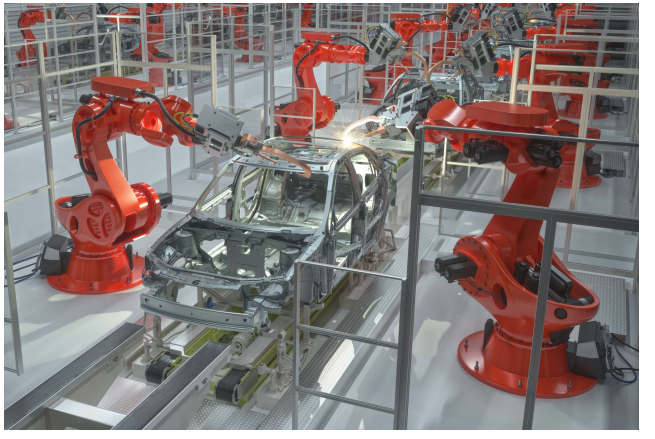
Take full control of routine tasks,
leave developer / tester / integrator
no chance for a mistake.

Automation is not a rocket science,
but it takes significant effort
to fit the moving parts,
make them robust and user-friendly,
and keep them running smoothly ever after

Automation requires discipline
and imposes constraints
on the workflows and the product

- automation does not tolerate
  - low quality components
  - complex workflows
  - rich option space

GettyImages-1211504544



GettyImages-105947848

"It's the little differences" (Pulp Fiction)

GettyImages-1396149748

Keep it simple: settle for the fit-for-purpose solutions

- focus on essentials
- minimize nice-to-haves
- routine tasks should require zero effort
- robust defaults are vital

Be defensive, shield your project from surprise factors:

- version all the components
- stage / test all your 3rd-party dependencies
- **automatically recover from system hiccups**
- **collect as much diagnostics as possible**

Quality enablers (think of UX …well, developer/tester experience):

- short build times (5min max for a full build)
- short test times (5min max for the entire test suite)
- smart regression comparators (minimize false-positives)

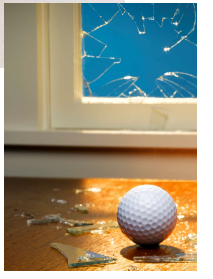
Floating Point arithmetic

Validating round-off errors is extremely hard:

- **no tolerance is safe**
  - binary match is the only safe indicator of no regression
  - all results must be deterministic and partition invariant
  - isolate/justify commits resulting in a drift
- -ffast-math switch is evil, IEEE compliance is critical
- modern compilers yield predictable FP results on CPUs
  - GPU compilers have space for improvement
- mixed/low-precision arithmetic is tempting for acceleration
  - can potentially give you a 2x boost
  - is sensitive to the code/math quality
  - does not tolerate ill-conditioned problems

GettyImages-826680374

Non-technical challenges



IT operations (system, build, integration, automation):

- **frequently under-appreciated**,
  mistakenly viewed to be secondary to coding
- **have hard-to-measure deliverables**
- **suffer from negative visibility** ("sysadmin curse"):
  - smooth operations are usually taken for granted
  - occasional screw-ups generate negative feedback
- **require conservative attitude**
- **require highly scattered unstructured subject knowledge**
  "They don't teach this stuff at school." (Alan Wild, HPC Systems)

Constant pressure to provide immediate solution
rather than a sustainable / manageable / scalable one

GettyImages-6577-000054
GettyImages-103586953

Ready for a run?

- Set the right expectations, manage software as a tangible product
- Give tough love: fight complexity, keep focus sharp, stay conservative
- Resist the hype of cutting-edge technologies,
  ”sustainable” is the opposite of ”fashionable” and ”experimental”
- Make post-release support and operations first class citizens
- Automate all routine tasks
  take control of little things
  before they take control of you

GettyImages-78377916