Getting to Adoption: Lessons from MPI and PETSc

William Gropp
wgropp.cs.Illlinois.edu

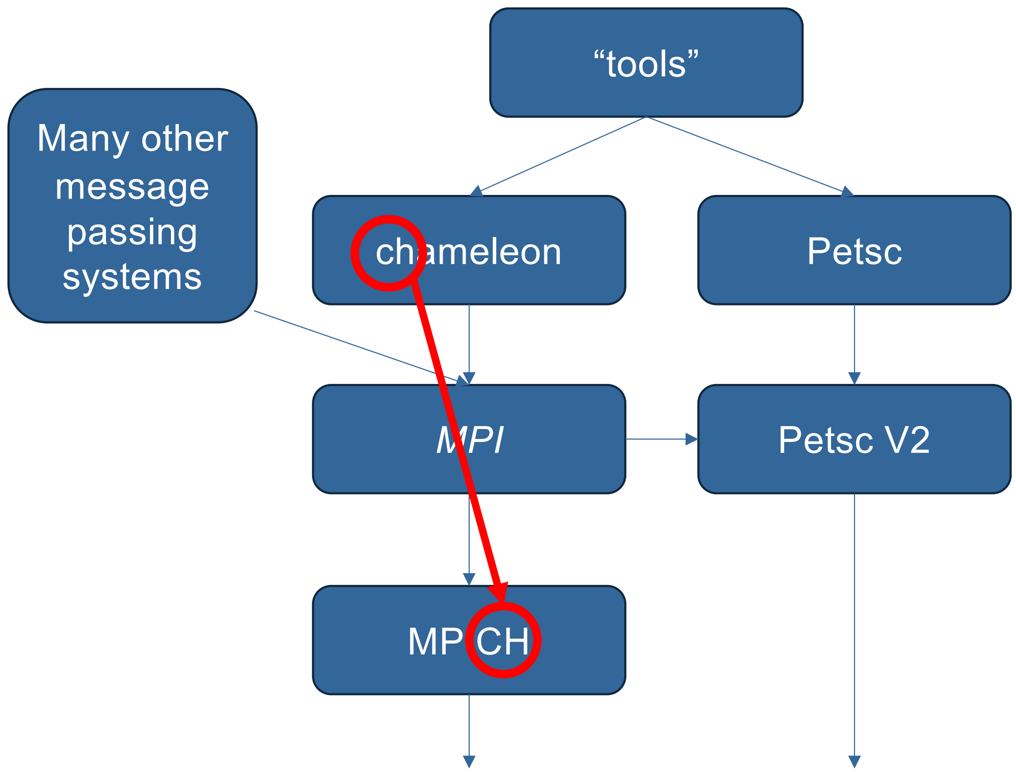
ILLINOIS NCSA

Background

- PETSc - grew from my own research project, focus on what a demanding user needed
- MPI - grew from the need to avoid gratuitous variations in APIs
- Both have been surprisingly successful
  - Both still in use nearly 30 years (!) later
- Why both MPI and PETSc
  - They aren't the same – MPI is a specification (standard) with multiple implementations, PETSc is a software package defined by what is implemented
  - What are you aiming for? Implementation? Specification? Both?

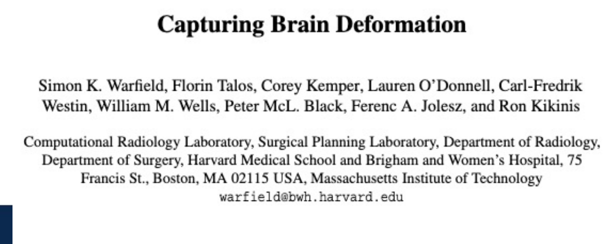
Two Entangled Projects

- Research into algorithms for domain decomposition with implicit methods for PDEs on distributed memory parallel computers
  - Like the iPSC d7
- No existing numerical library provided the necessary functionality
  - Anyone remember “reverse communication?
- Like everyone else, deal with lack of standard programming system for the many parallel computers



Common features

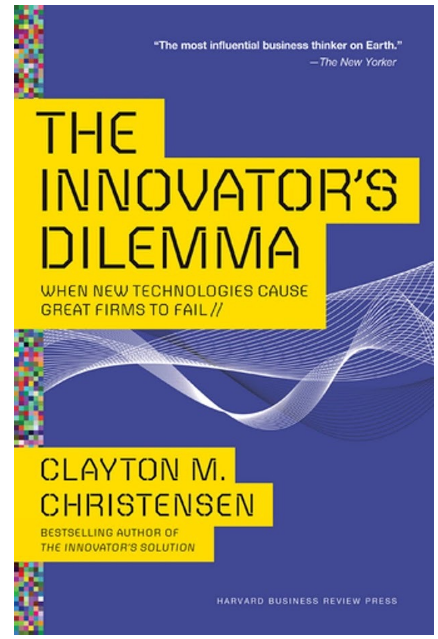
- The adoption of PETSc (software) and MPI (specification) have many common features
  - Next five slides
  - The obvious one I'm not going to talk about – good design
  - Less obvious - Luck
- Adoption of new technology can be disruptive
  - Conditions for success often similar
  - Not enough to be a little better/cheaper
  - Best is new capabilities, unexpected uses



Capturing Brain Deformation

Simon K. Warfield, Florin Talos, Corey Kemper, Lauren O'Donnell, Carl-Fredrik Westin, William M. Wells, Peter McL. Black, Ferenc A. Jolesz, and Ron Kikinis

Computational Radiology Laboratory, Surgical Planning Laboratory, Department of Radiology, Department of Surgery, Harvard Medical School and Brigham and Women's Hospital, 75 Francis St., Boston, MA 02115 USA, Massachusetts Institute of Technology
warfield@bwh.harvard.edu



Designed around users

- PETSc was not conceived as a numerical library
  - I needed a more flexible set of numerical routines – one that cleanly separated the algorithm and data structure from the mathematical operation
  - It turned out that I wasn't the only one that needed this
- Having a **specific** set of application needs was key in ensuring the design of PETSc met user needs

- MPI history
  - Ken Kennedy wanted a reliable layer on which to implement higher level parallel programming models, e.g., HPF
  - There were many different message passing systems – too many
  - MPI Forum used an open process, with vendors, researchers, and users represented
  - https://wgropp.cs.illinois.edu/bib/talks/tdata/2023/MPI-Lusk.pdf

Aggressively portable

- PETSc ran on a wide variety of environments, even non-Unix
- Several generations of build systems
- Early adopter of capability based portability
  - E.g., HAS_AIO, not AIX (or system name)
- Follow language standards (avoid compiler extensions, no matter how inviting)
  - Or isolate use to enable workarounds

- MPI designed to be OS agnostic; implementations ran on wide variety of systems (even non-Unix)
- Greatest Common Denominator approach
- Abstraction hides network and implementation specifics
- Early implementations also strived for portability
  - MPICH designed to allow incremental implementation; start with a run-everywhere model then add extra-value features, such as collectives in network
  - Many vendors able to use MPICH as the basis for their own MPI

Sweet spot of portability, performance, and generality

- PETSC provides end-to-end support for the user's application
  - Both high-level, easy to use routines and lower-level, easier to optimize, routines.
  - Similar philosophy given as one reason for the success of Python (IEEE Spectrum 9/23 p 2)
- Attention to performance
  - Manually unrolled loops when compiler wouldn't
  - New "multivector" routines to reduce memory motion
  - Performance analysis guided tuning; "achievable performance" in 1999 Gordon Bell winning application foreshadowed roofline analysis

- MPI – Early focus on performance (no extra memory copies, which was not a feature of some alternatives)
- MPI may be both too high-level and too low-level at the same time – but maybe that is one reason that it succeeded?
  - https://www.mcs.anl.gov/mpi-symposium/slides/marc_snir_25yrs mpi.pdf
- There are many reason for MPI's success; see "Learning from the Success of MPI"
  - https://link.springer.com/chapter/10.1007/3-540-45307-5_8

Development of documentation and training

- PETSC
  - BYOC (bring your own code) workshops (we'd call them hackathons today).
  - Many example codes, including typical use for PDE solutions, distributed with software
  - Documentation for all routines
    - With some ***content***, not just regurgitating the definition in the code
    - You have to write documentation, not just extract from the code definition
- MPI
  - Many people have done tutorials from 1-5 days
  - Documentation (see PETSc)
  - Books – Including Using MPI (now in 3rd edition), Using MPI2, and Using Advanced MPI
  - Some example codes (though not, frankly, at the level of PETSc's)

User support

- PETSc takes bug reports and often acts on them
  - Not only for implementation bugs but for desired features
  - Formal bug tracking
    - Several generations of tools
- MPI Forum takes user comments
  - Ex. An early meeting before MPI-1 released took user feedback and added a few functions (buffered send and wtime/wtick)
- MPI Implementations have active communities including bug reporting

Special features of a specification

- You can't change things on a whim.
  - Take more time to get it right the first time
  - Does slow down innovation
  - But does mean that you can build tools without version hell
  - (Containers are a pragmatic answer to the failure of software engineering)
- Benefits include the amortization of comprehensive documentation and examples over longer time because of the stability of the definition
- Separation of specification from implementation encourages abstraction

Other Notes

- Implementations
  - You get one chance with most users (even me!)
  - Gratuitous differences and changes are a barrier as well as extra work
  - If performance is important, make sure you deliver
    - You'll need to know what performance is achievable
  - If scalability is important, make sure you deliver
    - Make sure you define you terms
      - Scalability to some is 4 cores in one socket; to others 1M cores in 10K nodes.

Adoption barriers

What happens if you go away? What is someone's fallback?

- PETSc
  - Open Source (so others could in principle pick up the code) + new capabilities (high level operators rather than explicit data structure/algorithm choices)
- MPI
  - Open Source implementations (plural, reduces risk) as well as commercial. As a specification, mostly codified existing practice (at least in the beginning) so that subsets mostly easily implemented.

Have an answer to "what happens if you go away?"

- Being commercial is no longer any guarantee – see the 293(!!) dead projects at https://killedbygoogle.com/ . If anything, a successful open-source project is less risky than a commercial project because (a subset of) the community can decide to maintain it.

Lessons

1. Understand and meet user need
    - Design for a well-understood audience - That might be yourself
    - Not “build it and they will come”
2. Run in user’s environment
3. Provide real value
    - Know your place in the ecosystem
        - End-to-end solutions are often easiest for the user
        - Second are “drop-in” replacements
    - Not “This works for benchmarks” – model problems often have simplicities not shared with problems of interest
4. Document – nothing is as obvious as you think
5. Provide support – someone will need to answer questions
6. Understand the concerns of your audience
    - What happens to them if you disappear?