Using Empirical Study to Learn about the Development of High-End Codes

Development Time Working Group of
High Productivity Computing Systems (HPCS) Project

Victor R. Basili
University of Maryland
and
Fraunhofer Center - Maryland

Outline

- Empirical software engineering
- Empirical software engineering in the HPCS domain
- Our research approach
- Example results
- Final thoughts

Setting the Context

• Software engineering is an *engineering* discipline

• We need to understand products, processes, and the relationship between them (*we assume there is one*)

• We need to *experiment* (human-based studies), analyze, and synthesize that knowledge

• We need to package (model) that knowledge for use and evolution

➔ Recognizing these needs **changes how we think**, what we do, what is important, and the nature of the discipline

Motivation for Empirical Software Engineering

Understanding a discipline involves

– Observation,	Gaining knowledge
– Model building,	Encapsulating knowledge
– Experimentation,	Checking knowledge is correct
– and Evolution.	Changing knowledge as we learn more

This is the empirical paradigm that has been used in many fields, e.g., physics, medicine, manufacturing

Empirical software engineering involves the scientific use of quantitative and qualitative data to understand and improve the software product, software development process and software management

In software engineering, this paradigm requires “real world laboratories.” Research and Development have a synergistic relationship that requires a working relationship between industry and academe

Motivation for Empirical Software Engineering

For example, a software organization needs to ask:

- What is the right combination of technical and managerial solutions for my problem and my environment?
- What are the right set of processes for that business?
- How should they be tailored?
- How do we learn from our successes and failures?
- How do we demonstrate sustained, measurable improvement?

More specifically in their particular environment:

- When are peer reviews more effective than functional testing?
- When is an agile approach appropriate?
- When do I buy rather than make my software product elements?

Examples of Useful Empirical Results

"***Under specified conditions***, …"

Technique Selection Guidance

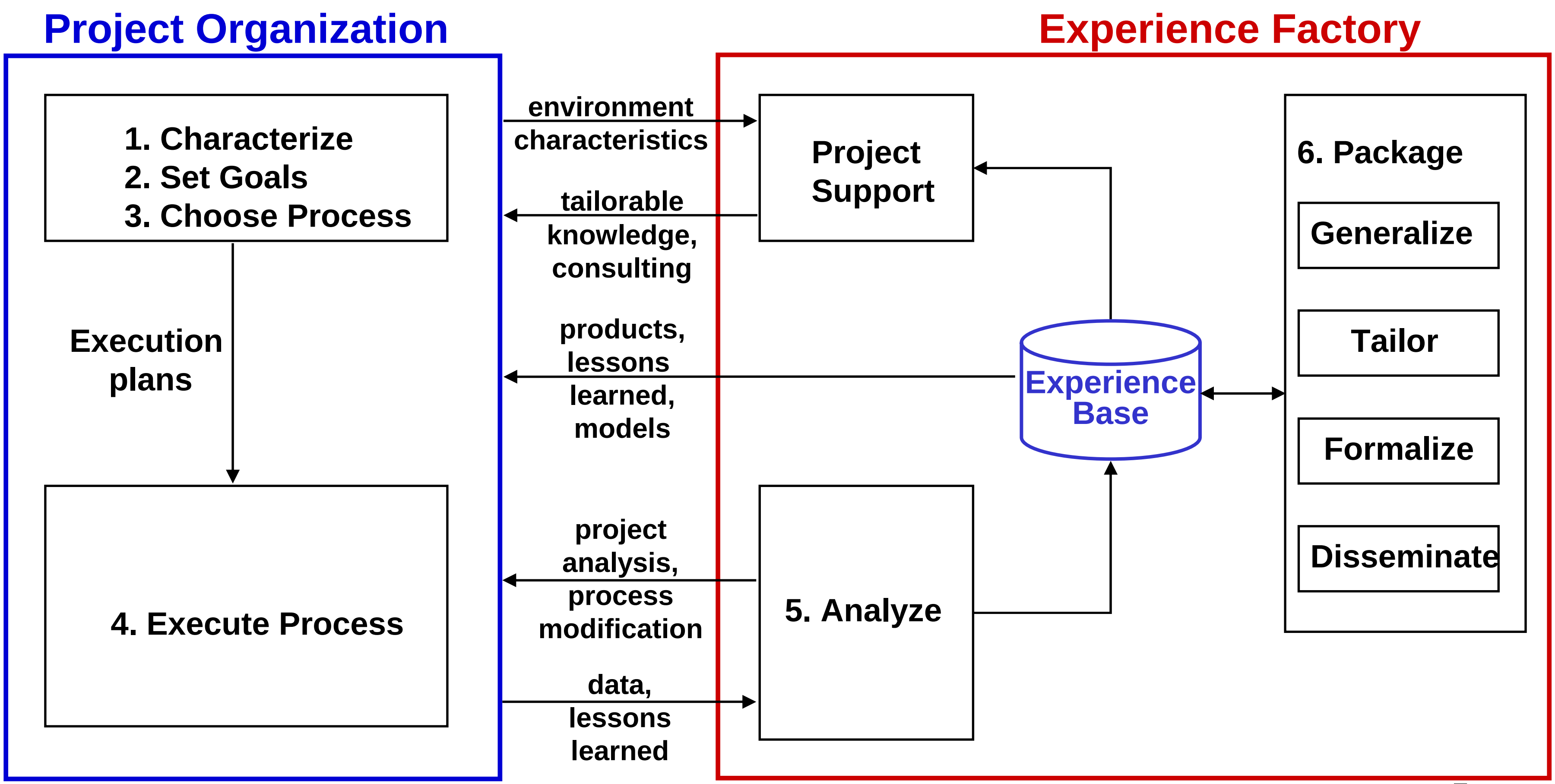
- Peer reviews are more effective than functional testing for faults of omission and incorrect specification
- Functional testing is more effective than reviews for faults related to numerical approximations and control flow

Technique Definition Guidance

- For a reviewer with an average experience level, a procedural approach to defect detection is more effective than a less procedural one.
- Procedural inspections, based upon specific goals, will find defects related to those goals, so inspections can be customized.
- Readers of a software artifact are more effective in uncovering defects when each uses a different and specific focus.

Basic Concepts
for Empirical Software Engineering

The Experience Factory implements learning cycles in software organizations by *building* software competencies and *supplying* them to projects.



The Experience Factory Organization
A Different Paradigm

Project Organization **Problem Solving**	**Experience Factory** **Experience Packaging**
Decomposition of a problem into simpler ones	Unification of different solutions and re-definition of the problem
Instantiation	Generalization, Formalization
Design/Implementation process	Analysis/Synthesis process
Validation and Verification	Experimentation
Product Delivery within Schedule and Cost	**Experience / Recommendations Delivery to Project**

An Example Experience Factory Structure

NASA Software Engineering Laboratory (SEL)

Used baselines to show improvement of ground support software for satellites

Three baselines: 1987 vs. 1991 vs. 1995

Continuous Improvement in the SEL:

Decreased **Development Defect rates** by
75% **(**87 - 91) **37%** (91 - 95)
Reduced **Cost** by
55% (87 - 91) **42%** (91 - 95)
Improved **Reuse** by
300% (87 - 91) **8%** (91 - 95)
Increased **Functionality** five-fold (76 - 92)

Outline

- Empirical software engineering
- Empirical software engineering in the HPCS domain
- Our research approach
- Example results
- Final thoughts

High Productivity Computing Systems (HPCS)

Problem: How do you build sufficient knowledge about the high end computing (HEC) so you can improve the time and cost of developing these codes?

Project Goal: Improve the buyer's ability to select the high end computer for the problems to be solved based upon productivity, where productivity means

Time to Solution = Development Time + Execution Time

Research Goal: Develop theories, hypotheses, and guidelines that allow us to characterize, evaluate, predict and improve how an HPC environment (hardware, software, human) affects the development of high end computing codes.

Partners: MIT Lincoln Labs, MIT, MSU, UCSD, UCSB, UCSD, UH, UMD, UNL, USC, FC-MD, ISU

HPCS Example Questions

- **How does a HEC environment (hardware, software, human) affect the development of an HEC program?**

  – What is the **cost** and **benefit** of applying a particular HPC technology (MPI, Open MP, UPC, Co-Array Fortran, XMTC, StarP,…)?

  – What are the **relationships** among the technologies, the work flows, development cost, the defects, and the performance?

  – What **context variables** affect the development cost and effectiveness of the technology in achieving its product goals?

  – Can we build **predictive models** of the above relationships?

  – What **tradeoffs** are possible?

  – …

HPCS Experience Packaging

Development Time Experiments – Novices and Experts

Empirical Data

Predictive Models

(Quantitative Guidance)

General Heuristics

(Qualitative Guidance)

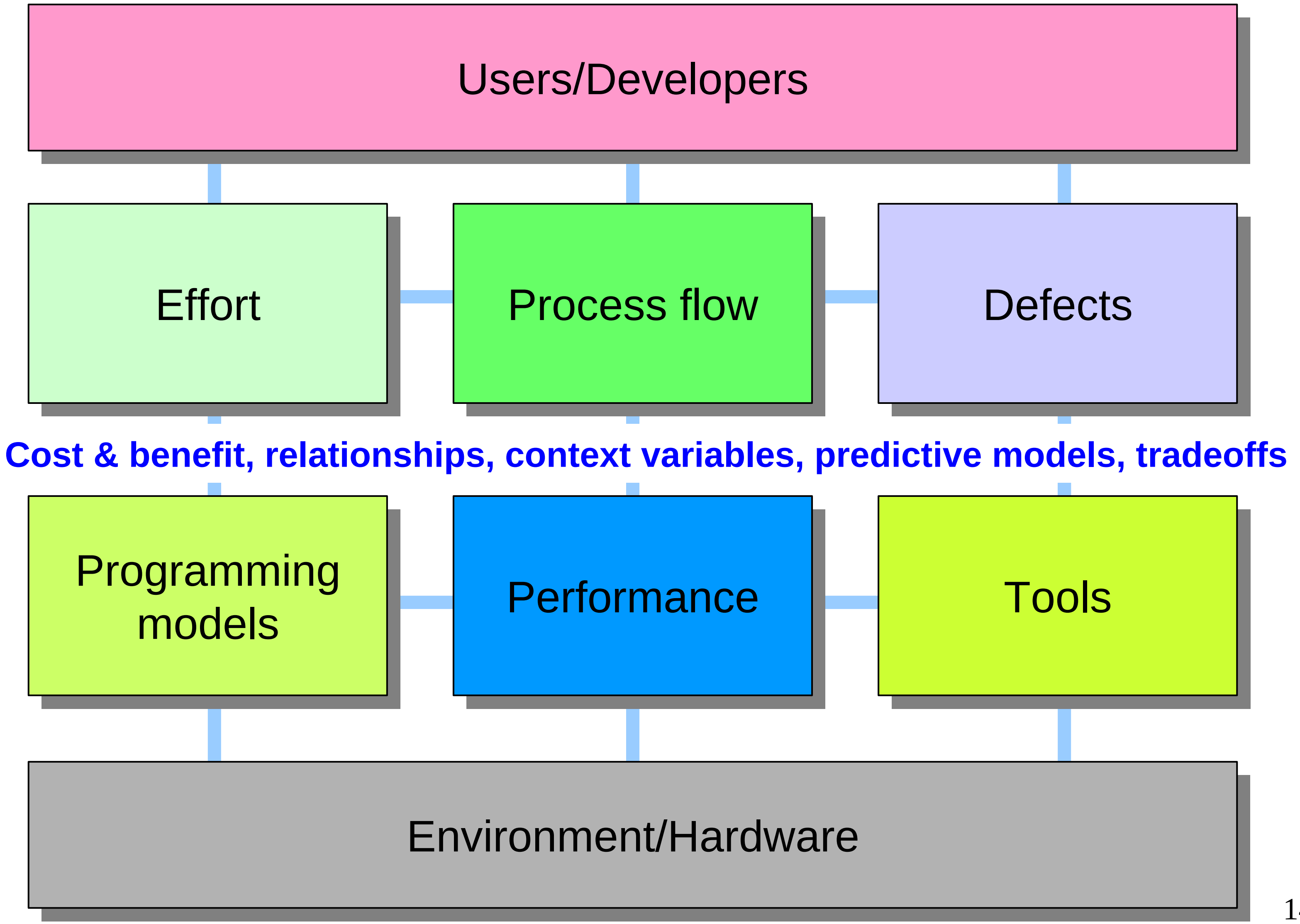
E.g. Tradeoff between effort and performance:

MPI will increase the development effort by y% and increase the performance z% over **OpenMP**

E.g. Experience:

Novices can achieve speed-up in cases X, Y, and Z, but not in cases A, B, C.

Areas of Study



Areas of Study

- **Effort**
  - How do you measure effort? What variables affect effort? Can we build and evolve hypotheses about the relationship between effort and other variables? Can we identify effective productivity variables, e.g., values and costs?

- **Process flow**
  - What is the normal process followed? What is the breakdown between work and rework? Can we use automated data collection to automatically measure process steps?

- **Defects**
  - What are the domain specific defect classes? Can we identify patterns, symptoms, causes, and potential cures and preventions? Can we measure effort to isolate and fix problems?

Outline

- Empirical software engineering
- Empirical software engineering in the HPCS domain
- Our research approach
- Example results
- Final thoughts

Types of Studies

Controlled experiments

Study programming in the small under controlled conditions to:
Identify key variables, check out methods for data collection, get professors interested in empiricism

E.g., compare effort required to develop code in MPI vs. OpenMP

Observational studies

Characterize in detail a realistic programming problem in realistic conditions to:
validate data collection tools and processes

E.g., build an accurate effort data model

Case studies and field studies

Study programming in the large under typical conditions

E.g., understand multi-programmer development workflow

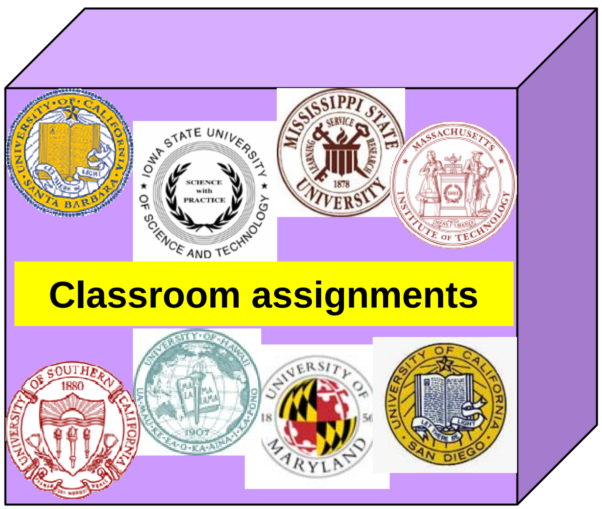
Surveys, interviews & focus groups

Collect “folklore” from practitioners in government, industry and academia

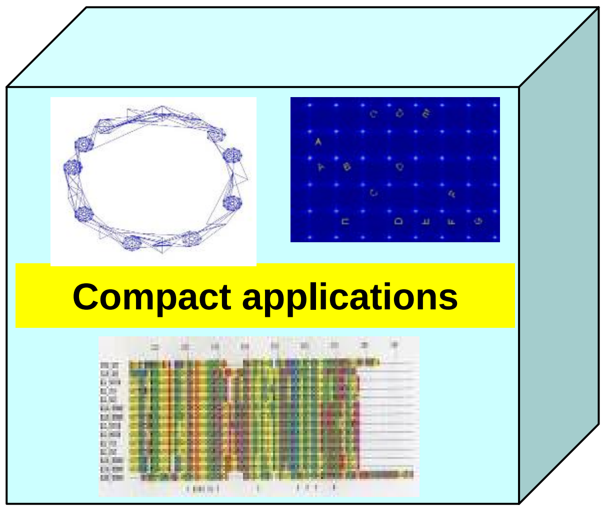
e.g., generate hypotheses to test in experiments and case studies

Types of Testbeds

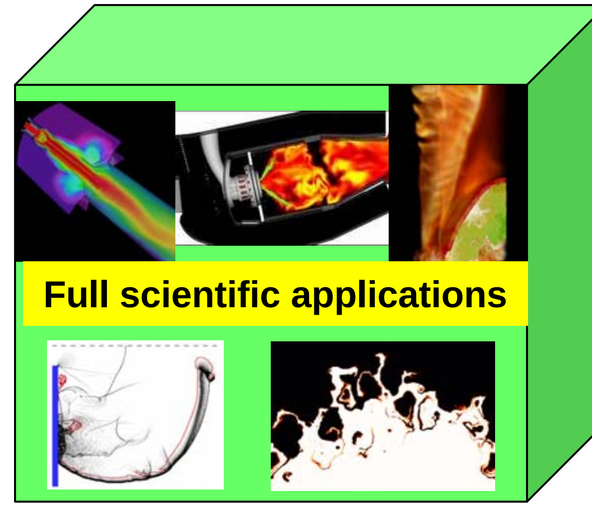
Experimenting with a series of testbeds ranging in size and perspective



Array Compaction, the Game of Life, Parallel Sorting, LU Decomposition,
Developed in graduate **courses** at a variety of universities



Bioinformatics, graph theory, sensor & I/O: combination of kernels, e.g., Embarrassingly Parallel, Coherence, Broadcast, Nearest Neighbor, Reduction
Developed by experts testing **key benchmarks**



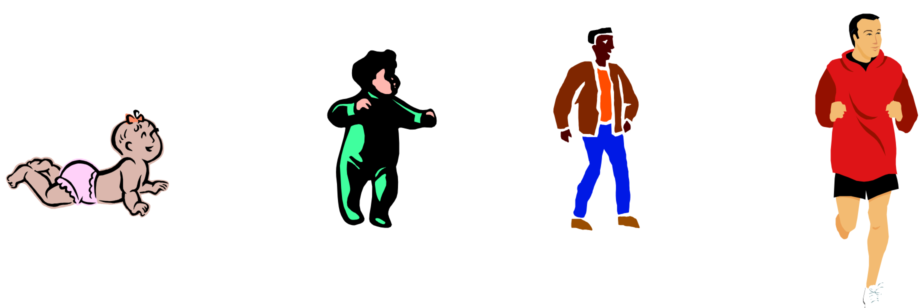
Nuclear simulation, climate modeling, protein folding, …
Developed at **ASCI Centers** at 5 universities
Run at the **San Diego Supercomputer Center**

Approach: Learning over time
Selecting studies and testbeds

- **Pilot controlled experiments** on **classroom assignments** (single programmer, graduate students)
  - Identify variables, data collection problems, workflows, experimental designs
- Lead to **observational studies** of **classroom assignments** (single programmers, graduate students)
  - Develop variables and data we can collect with confidence based

Crawl before you walk before you run

- Le… …**ents** (single programmers)
  - Generate more confidence in the variables, data collection, models, provide hypotheses about novices
- Lead to **case studies** of **classroom assignments** (teams)
  - Study scale-up, multi-developer workflows,
- Lead to **case studies** of **compact apps** (professional developers)
  - Study scale-up, multi-developer workflows,
- **Interviews** with developers and users in a variety of environments…



Approach: Learning over time Analysis and Synthesis

- **Identify relevant variables**, context variables, programmer workflows, mechanisms for identifying variables and relationships
  - Developers: Novice, experts
  - Problem spaces: various kernels; computationally- based vs. communication based; …
  - Work-flows: single programmer research model, …
  - Mechanisms: controlled experiments, folklore elicitation, case studies

- **Identify measures and proxies** for those variables that can be collected accurately or what proxies can be substituted for those variables, understand the data collection problems,

- **Identify the relationships** among those variables, and the contexts in which those relationships are true

- **Build models** of time to development, productivity, relative effectiveness of different programming models,
  - E.g., OpenMP offers more speedup for novices in a shorter amount of time when the problem is more computationally-based than communication based.

Approach: Learning over time
Formalizing results

- **Identify folklore***: elicit expert opinion to identify the relevant variables and terminology, some simple relationships among variables, looking for consensus or disagreement

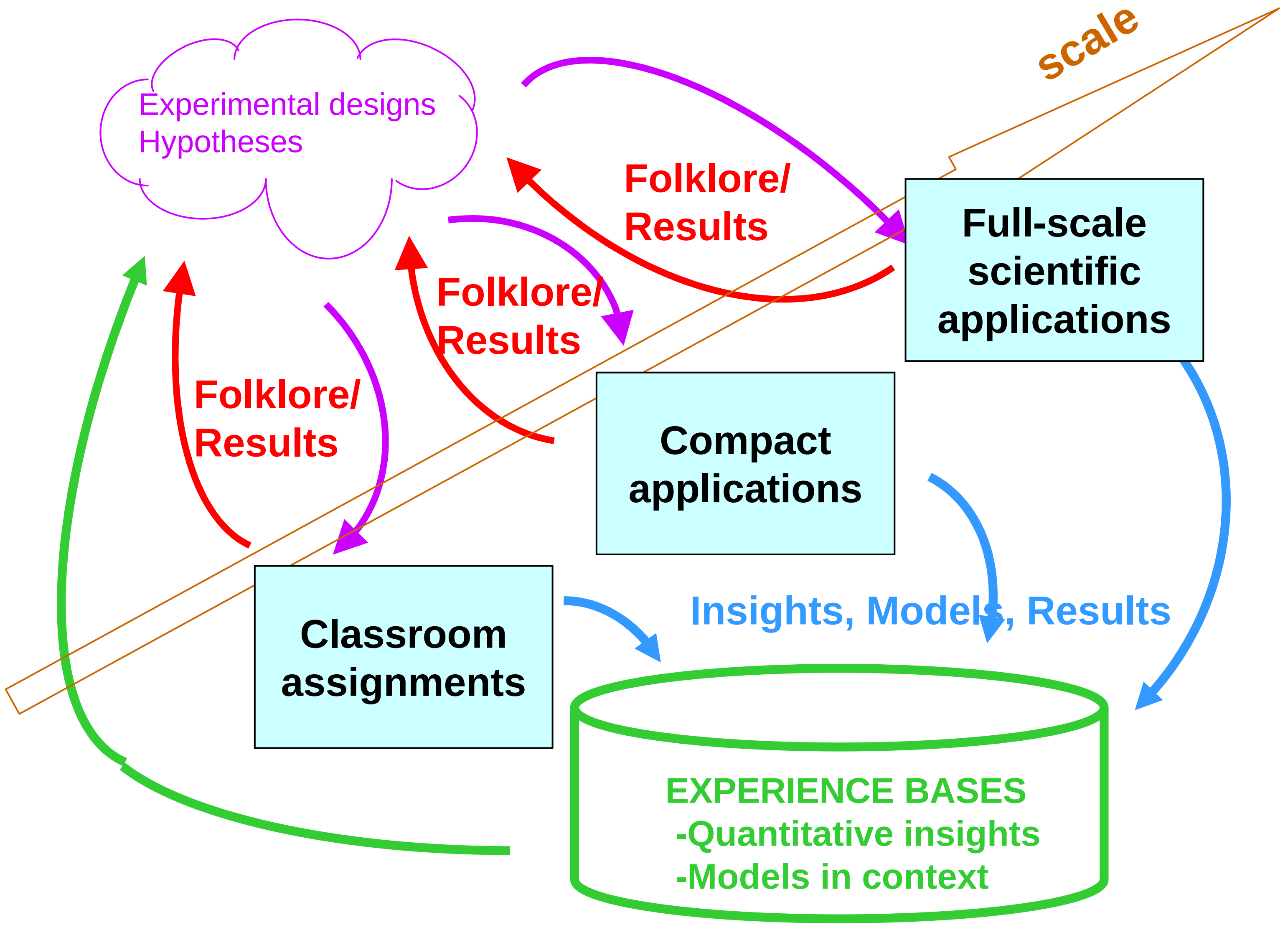
- **Evolve the folklore**: evolve the relationships and identify the context variables that affect their validity, using surveys and other mechanisms

- **Turn the folklore into hypotheses** using variables that can be specified and measured

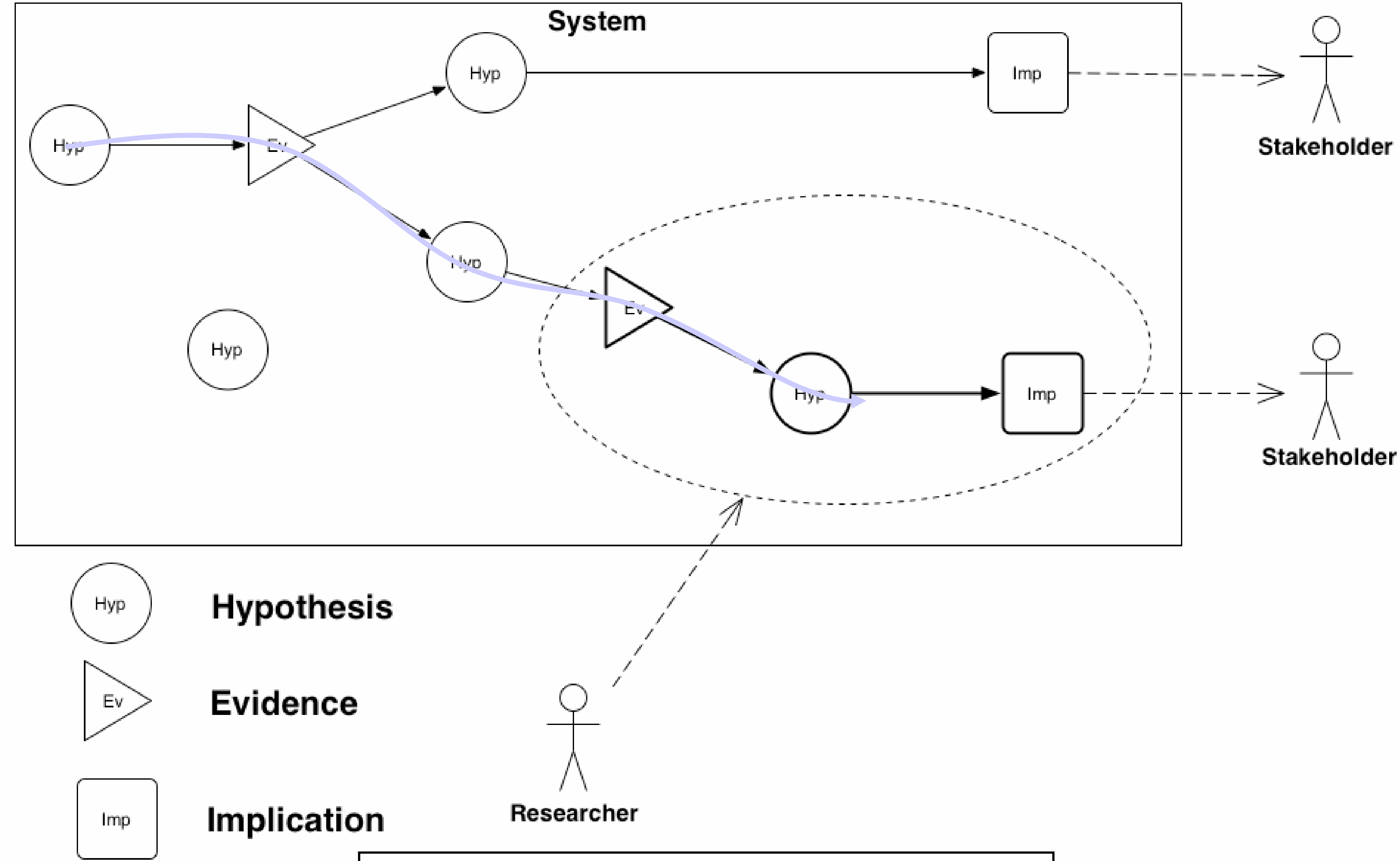
- **Verify hypotheses** or generate more confidence in their usefulness in various studies about development, productivity, relative effectiveness of different programming models,
  - E.g., Usually, the first parallel implementation of a code is slower than its serial counterpart.

*__Folklore__: An unsupported notion, story, or saying widely circulated

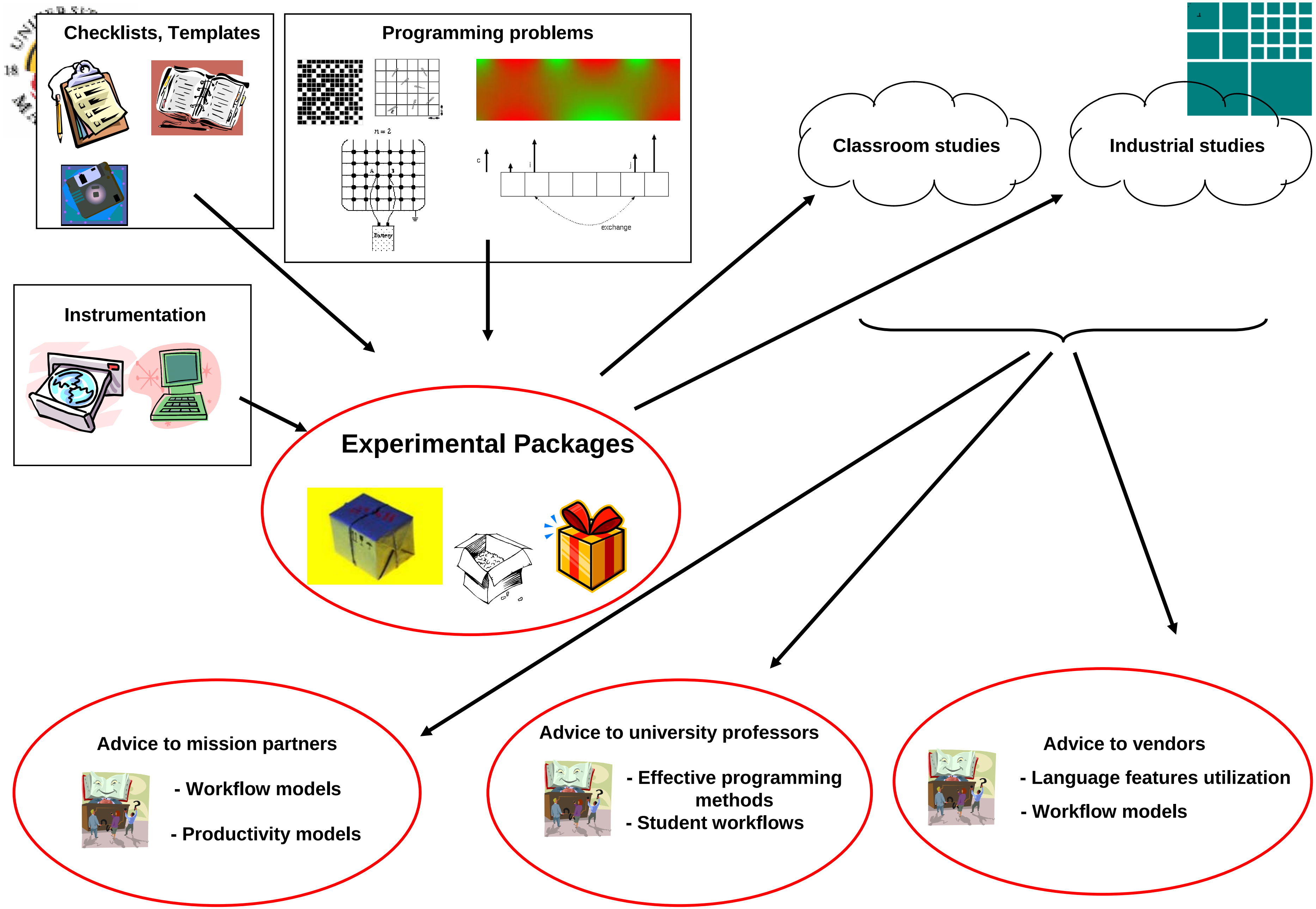
Building Experience Bases



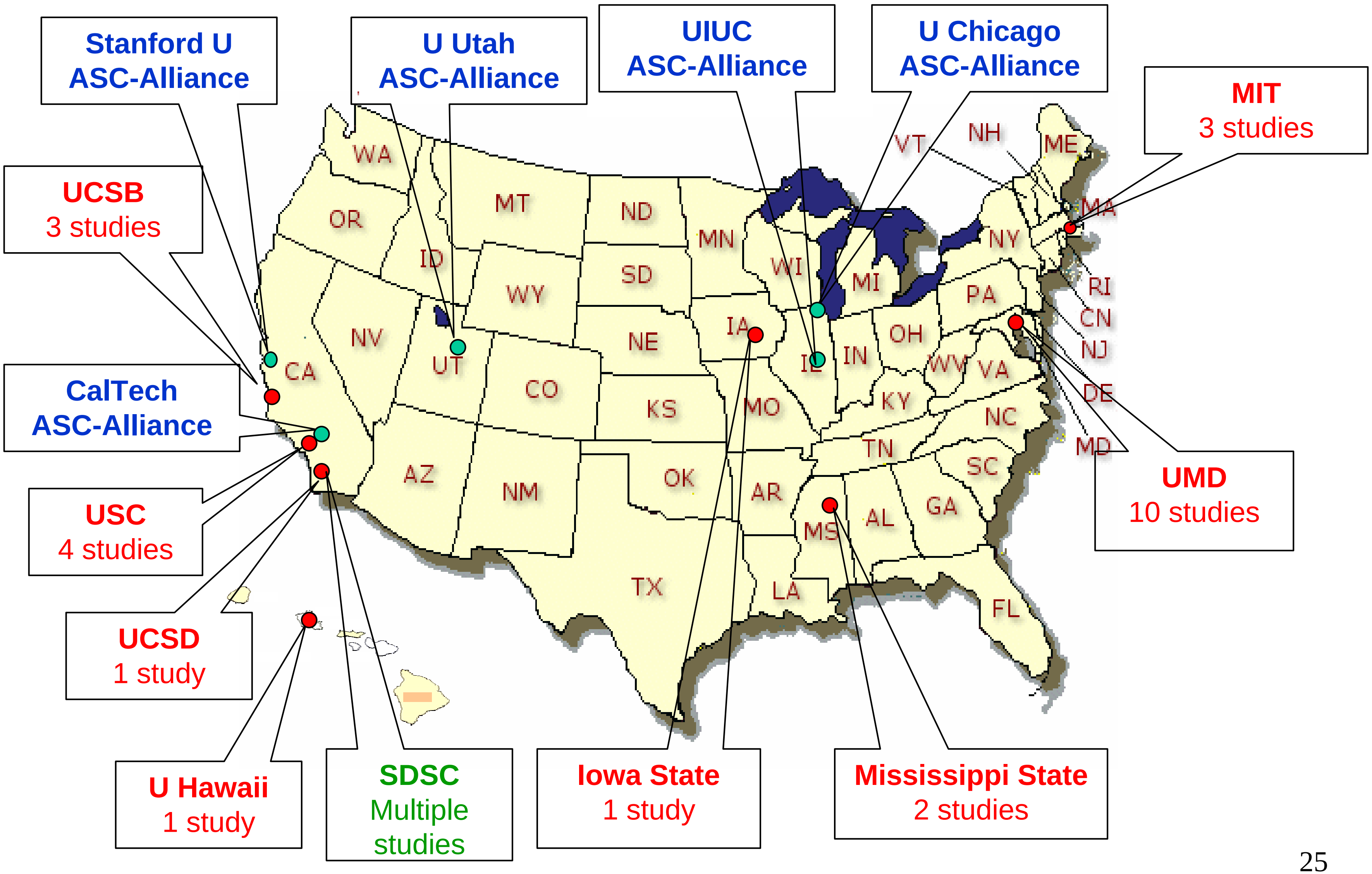
Building Experience Bases
Hypotheses, Evidence, Implications



Build a chain of evidence

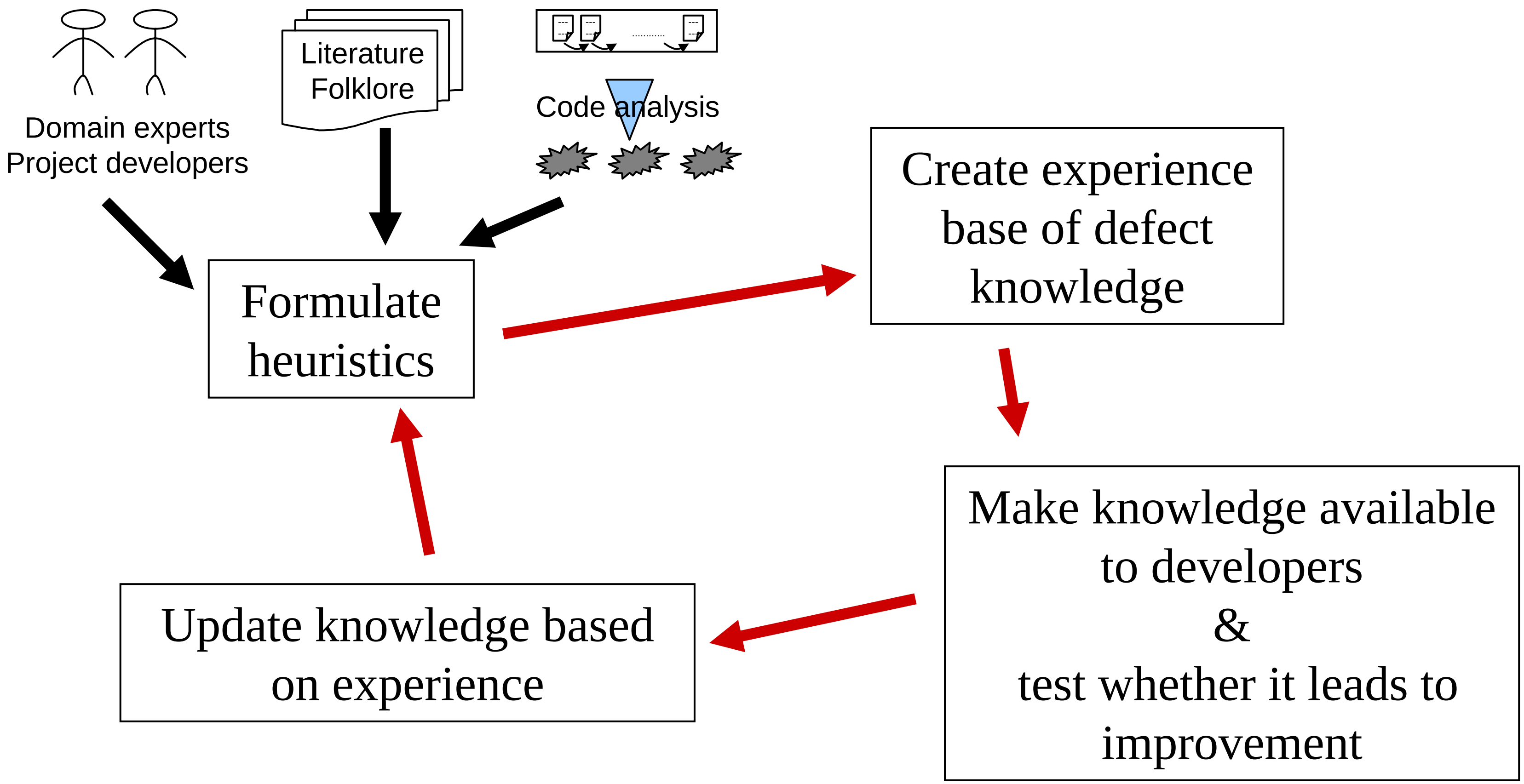

Checklists, Templates
Programming problems
Instrumentation
Classroom studies
Industrial studies
Experimental Packages
Advice to mission partners
- Workflow models
- Productivity models
Advice to university professors
- Effective programming methods
- Student workflows
Advice to vendors
- Language features utilization
- Workflow models

Approach: Multiple collaborations to generate necessary data



Example of our Approach: Bringing it all together

- **Building knowledge about defects**
  - Goal: Provide better guidance about the types of defects likely to occur during HEC software development
  - Hypothesis: Knowledge about historic defects common in the domain can help developers avoid them in the future.



Outline

- Empirical software engineering
- Empirical software engineering in the HPCS domain
- Our research approach
- Example results
- Final thoughts

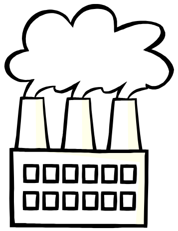
Results:
Infrastructure Tools & Packages

Experimenters' checklist

A checklist for professors and experts running studies. Includes **templates**, **forms**, and reusable **project artifacts**.

Value: Decreases effort for experimenters & increases validity of data comparisons across studies
http://care.cs.umd.edu:8080/hpcs/faculty/

HPCS Web Portal @ UMD
http://care.cs.umd.edu:8080/hpcs/



Downloadable instrumentation package for individual study & classroom study

Value: Once installed, allows minimally intrusive data collection and common definitions of the measures collected
http://care.cs.umd.edu:8080/hpcs/software/umdinst/

Experiment Manager

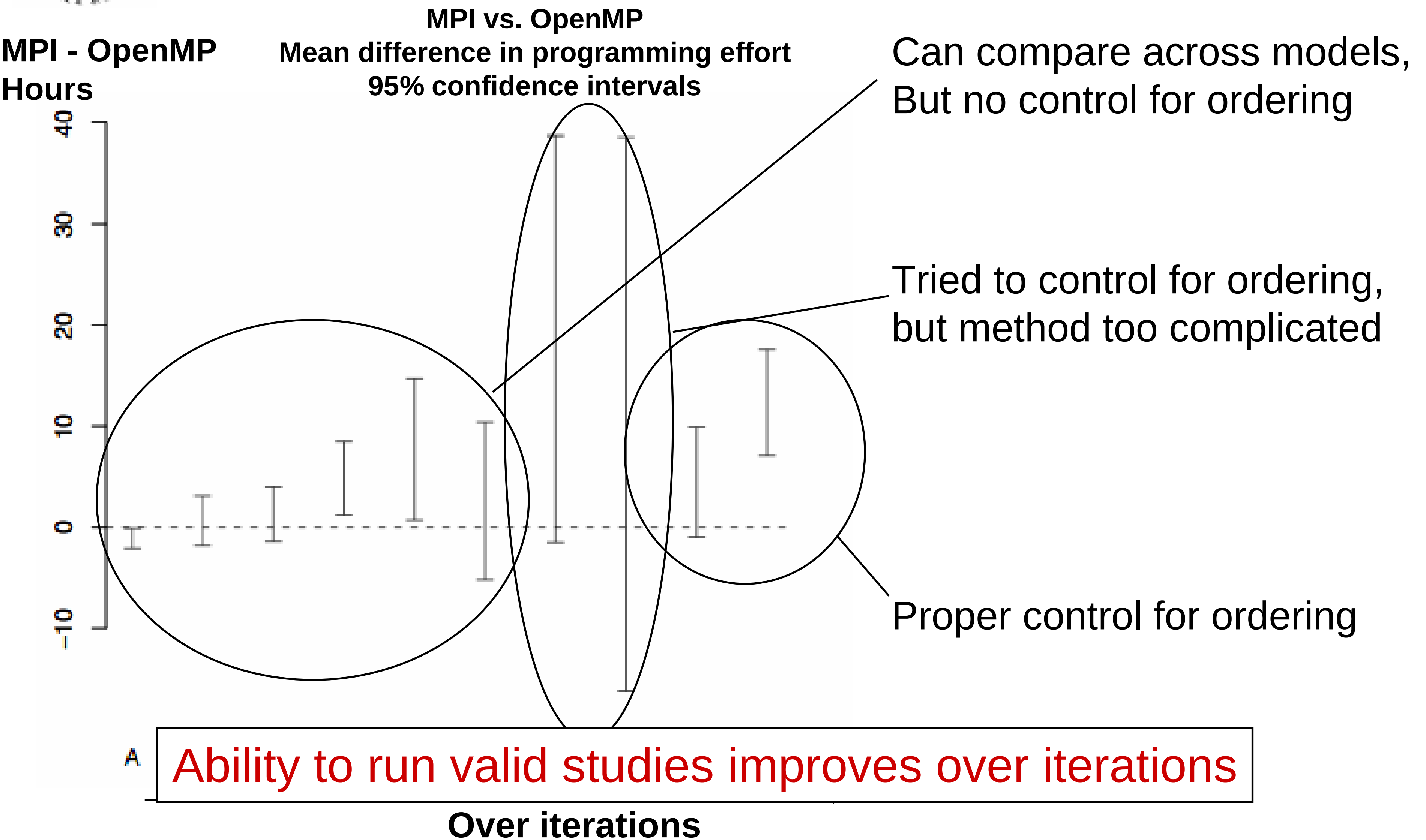
Web-based data repository

Value: Web-based front-end makes data collection require less effort
Subjects can send data directly to analysis team, doesn't require instructor/TA to be involved
Easy view of whether all students are contributing data
http://care.cs.umd.edu:8080/umdexpt/cgi-bin/index.cgi

Results: Accumulating Data Sets
(Controlled experiments, classroom assigns.)

Problem	serial	MPI	OpenMP	Matlab*P	XMT-C	Co-Array Fortran	UPC	Hybrid MPI-OpenMP
Game of life	4	5	2	1		2	2	
SWIM			1					
Buffon-Laplace	2	3	2	3				
Laplace's equation	1	1	1	1				
Sharks & fishes	1	2	2			1		
Grid of resistors	1	1	1	1				
Matrix power via prefix		3	1			1	1	
Sparse conjugate-gradient		2				1	1	
Dense matrix-vector multiply	1	1	1					
Sparse matrix-vector multiply	1	1			2			
Sorting	2	3	1		2			
Quantum dynamics		2						
Molecular dynamics								1
Randomized selection					1			
Breadth-first search					1			
LU decomposition			1					
Shortest path			1					
Search for intelligent puzzles		1						

Results: Comparing MPI & OpenMP
(Controlled experiments, classroom assigns.)

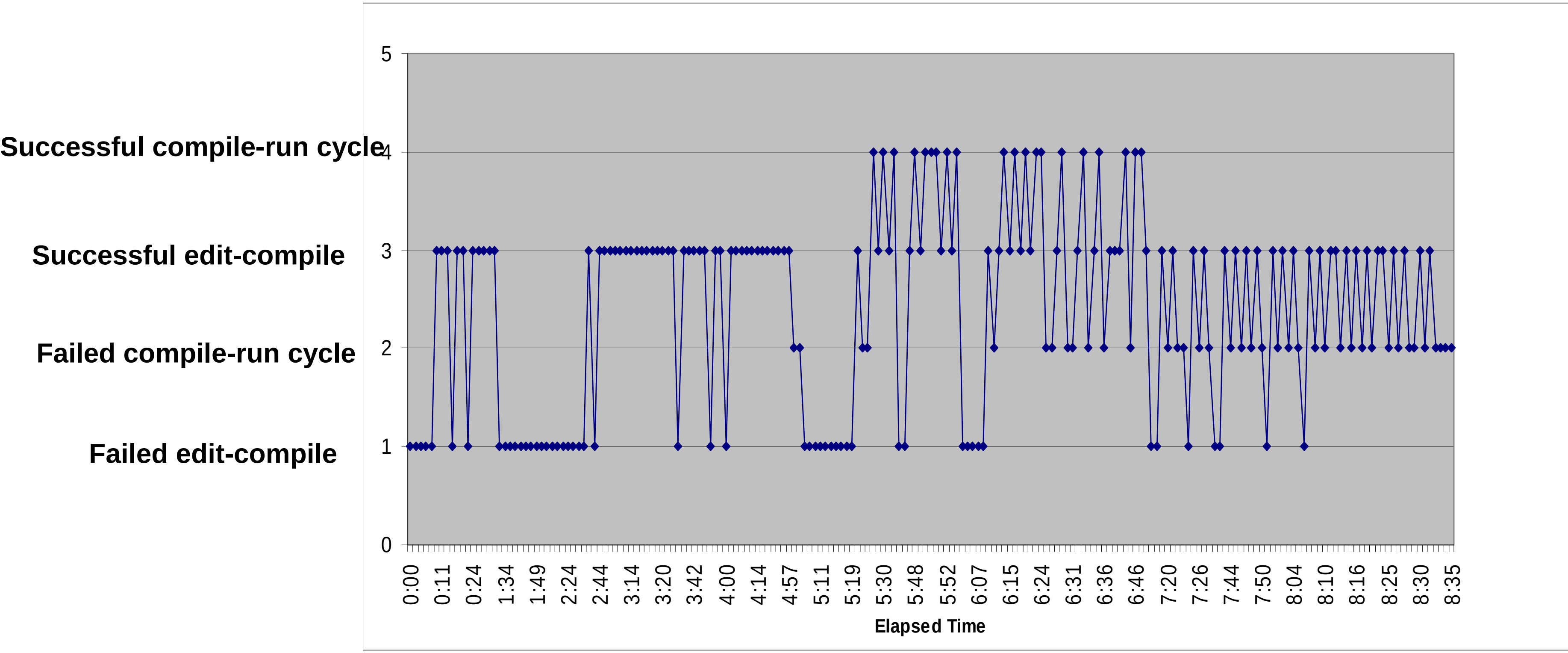


Ability to run valid studies improves over iterations

Results: Characterizing novices
(Synthesizing classroom assignments)

- OpenMP saves 35-75% of effort vs. MPI on most problems
- UPC/CAF saves ~40% of effort vs. MPI
- XMT-C saves ~50% of effort vs. MPI
- Experience with problem reduces effort, but effect of programming model is greater than effect of experience
- When performance is the goal:
  - Experts and students spend the same amount of time
  - Experts get significantly better performance
- Performance variation is considerable, especially for MPI
- Many do not achieve good performance
- No correlation between effort and performance

Results: Understanding workflow
(Observational study)



Observation:	A series of failed and successful Compile cycles with no runs	A series of failed and successful Compile-Run cycles	A series of successful Compile and failed Run cycles
Conclusion:	New code is being added and CompileTime defects being fixed	RunTime defects being fixed	Developer is not able to fix the defects

Results: Characterizing Processes
(Full-scale apps: SDSC, ASC)

- **Users** fall into **different categories**
  - Marquee users (run at very large scale, often using full system)
    - Often have a consultant to help them improve performance
  - Normal users (typically use 128-512 processors)
    - Less likely to need to tune
  - Small users (often novices just learning parallel programming)
- **Determining inputs** can take weeks, are themselves research projects
  - Modeling complex objects (e.g. space shuttle)
  - Determining initial conditions (e.g. supernova)
- **Debugging** is very challenging
  - Modules may work in isolation, but fail when connected together
  - Program may work on 32 processors, break on 64 processors
  - Hard to debug failures on hundreds of processors (print statements don't scale up!)
- **Visualization** is regularly used for validation
- Many projects have **no** one with a **computer science background**

Results: Characterizing Processes
(Full-scale apps: SDSC, ASC)

- **Performance** is treated as a **constraint**, not a **goal to be maximized**
  - Performance is important until it is “good enough” for their machine allocation
- **Portability** is a **must**
  - Can’t commit to technologies unless they know they will be there on future platforms
  - Some projects have broken compilers and libraries on every platform!
- Many users prefer **not** to **use performance tools**
  - Problems scaling to large processors
  - Difficult-to-use interfaces
  - Steep learning curve
  - Too much detail provided by tool
- **Codes are multi-language** and run on remote machines
  - Many software tools won’t work in this environment
- There is **extensive reuse of libraries**, but no reuse of frameworks
  - Everyone has to write MPI code

Results: Defect Knowledge
(Classification scheme abstracted from data)

Type	Sub-type	Description
Algorithm	---	Logical error
Side-effect of parallelization	File I/O Random function	Serial constructs causing correctness and performance defects when accessed in parallel contexts
Erroneous use of language features	---	Erroneous use of parallel language features
Space decomposition	---	Incorrect mapping between the problem space and the program memory space
Synchronization	Deadlock Race	Incorrect/unnecessary synchronization
Performance	Load balancing Scheduling	Scalability problem because processors are not working in parallel

Results: Defect Knowledge
(Example defect type description)

Pattern: Erroneous use of language features

- Simple mistakes in understanding that are common for novices
  - E.g., inconsistent parameter types between send and recv,
  - E.g., forgotten mandatory function calls
  - E.g., inappropriate choice of functions

Symptoms:

- Compile-type error (easy to fix)
- Some defects may surface only under specific conditions
  - (number of processors, value of input, hardware/software environment…)

Causes:

- Lack of experience with the syntax and semantics of new language features

Cures & preventions:

- Check unfamiliar language features carefully

Outline

- Empirical software engineering
- Empirical software engineering in the HPCS domain
- Our research approach
- Example results
- Final thoughts

An Operational Experience Base
(Defect Patterns, Symptoms, Causes, Cures)

The bases we are building have no worth without a community of users.
We invite you to visit!

Insights from experts

HPCBugBase (experience base)

http://care.cs.umd.edu:8080/hpcs/bugbase/

Feedback

Applications

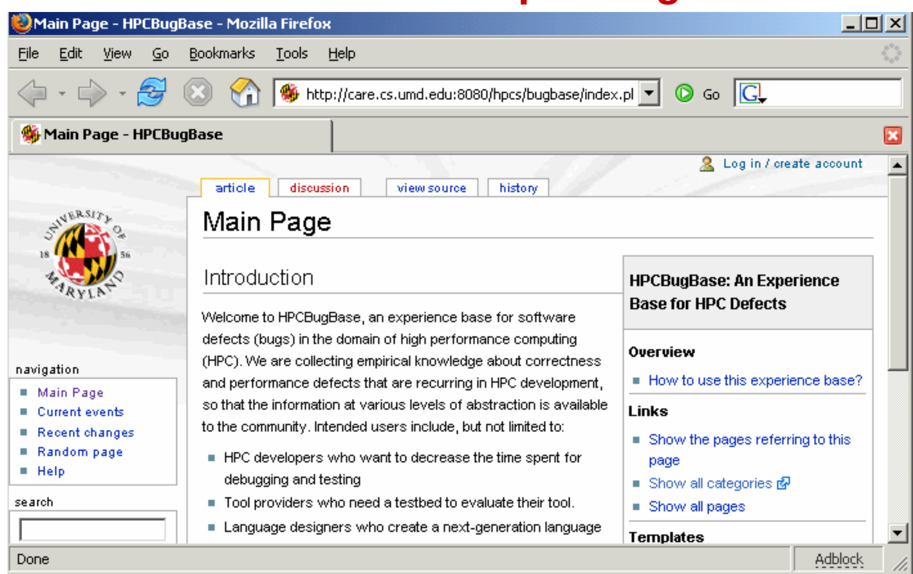
Training materials

Testbed for tools

Recommendations to technology providers

Analysis method

Assist analysis



Packaged knowledge

Data analysis

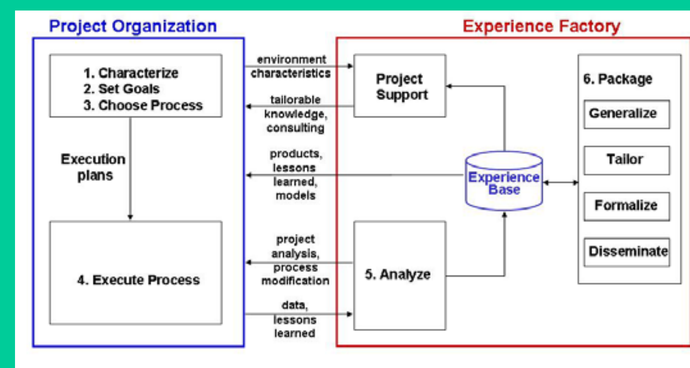
Source code history

Bug tracking systems
Mailing lists

Surveys/interviews

Document recurring correctness/performance problems at various levels of abstraction (source code, defect descriptions, advice, classification schemes)

What we believe



Software Engineering

Software Engineering is “**big science**”;
not small independent technology developments

Thanks to...

Study team:
UMD: Vic Basili, Marv Zelkowitz, Jeff Hollingsworth, Taiga Nakamura, Sima Asgari, Forrest Shull, Nico Zazworka, Rola Alameh, Daniela Suares Cruzes
UNL: Lorin Hochstein
MSU: Jeff Carver
UH: Philip Johnson
SDSC: Nicole Wolter, Michael McCracken

Professors teaching classes:
Alan Edelman [**MIT**], John Gilbert [**UCSB**], Mary Hall, Aiichiro Nakano, Jackie Chame [**USC**] Allan Snavely [**UCSD**], Alan Sussman, Uzi Vishkin, [**UMD**], Ed Luke [**MSU**], Henri Casanova [**UH**], Glenn Luecke [**ISU**]