# From Measuring Programs to Measuring Programmers

Jeffrey K. Hollingsworth

University of Maryland

## Background

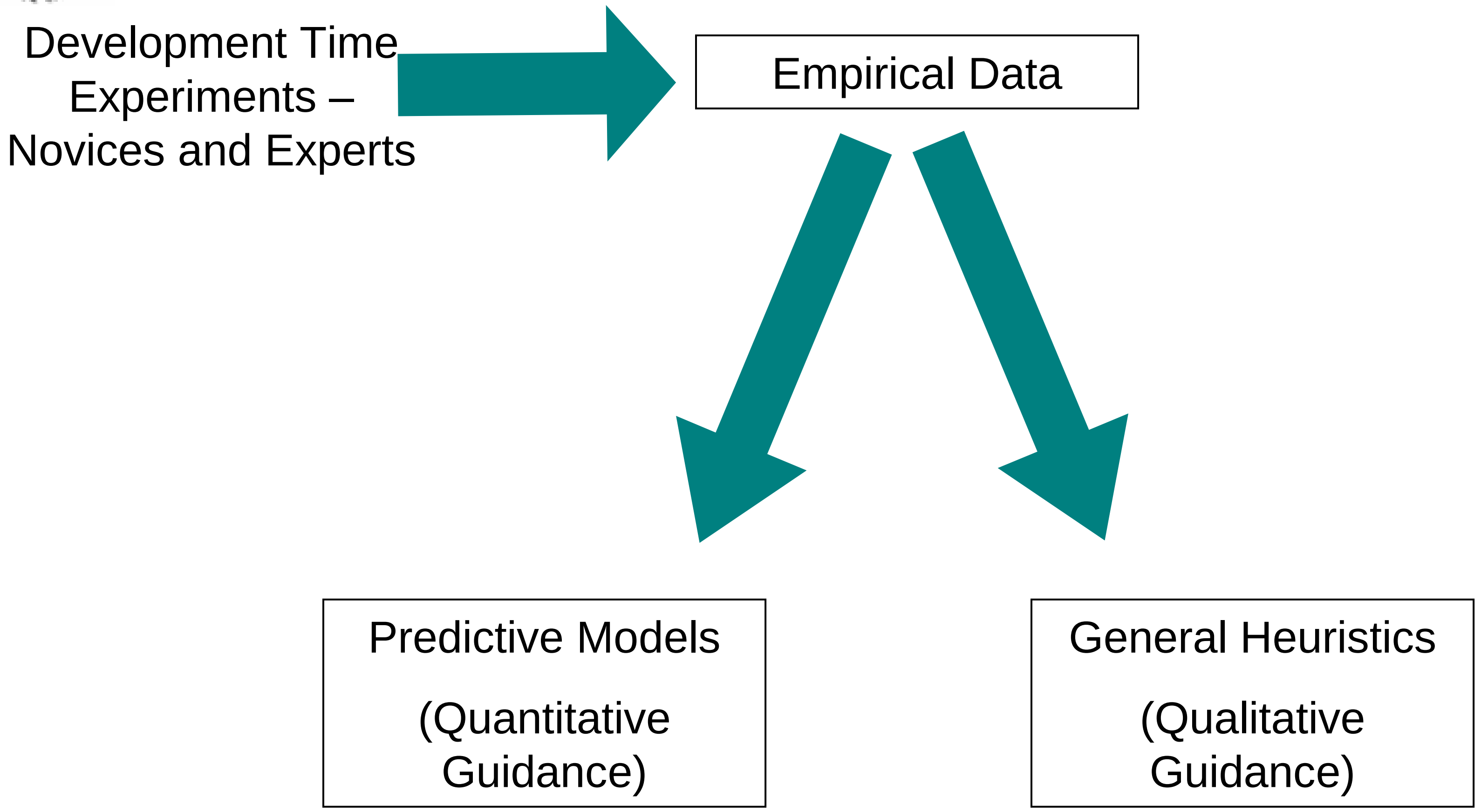
- Spent past 15+ years developing tools for tuning programs
  - Goal: Provide Information to get most out of machine
  - Assumption: Big machines are scarce, programmers do (and must) spend time getting every cycle available

- Recently (past 3 years) looking at programmer performance
  - Goal: Identify ways to get most out of programmers

## HPCS Example Questions

- **How does the HPC environment (hardware, software, human) affect the development of codes?**
  - 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 Research Activities


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.

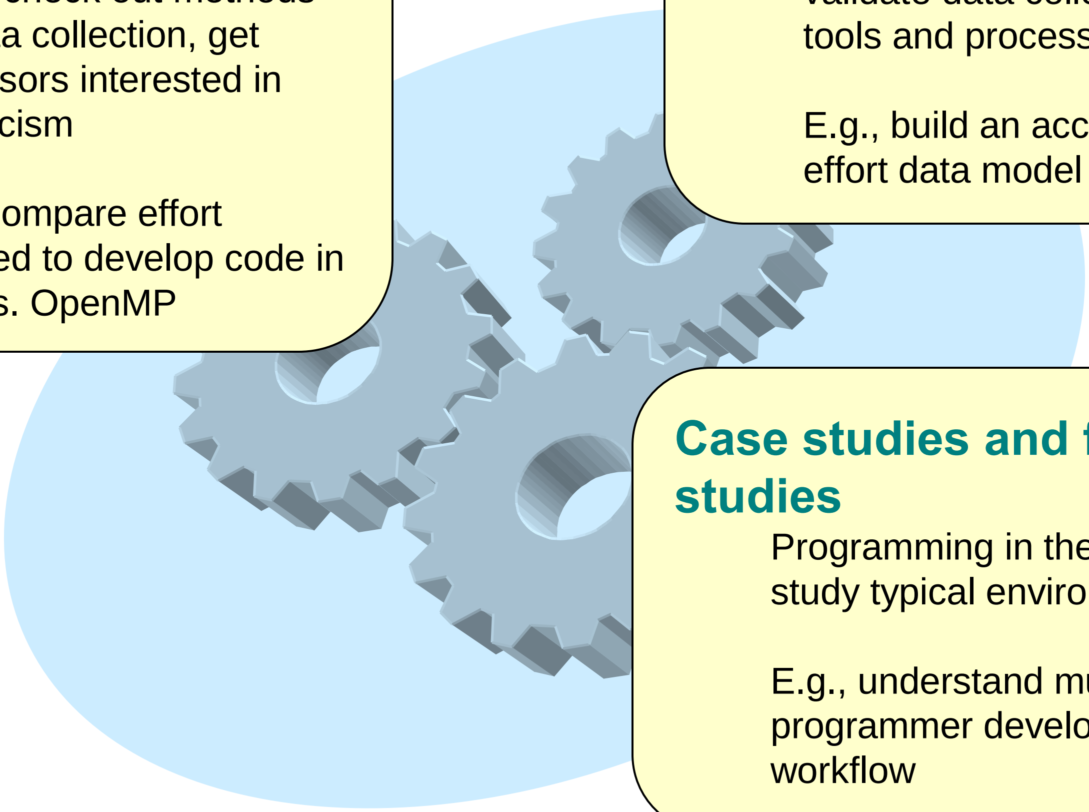
## Areas of Studies

- **Effort**
  - How do you measure effort? What variables effect 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?

- **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?

- **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?

## Type of Studies



### Controlled experiments

Identify key variables, study programming in the small, 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

Simulate the effects of the treatment variables in a realistic environment, validate data collection tools and processes

E.g., build an accurate effort data model

### Case studies and field studies

Programming in the large, study typical environments

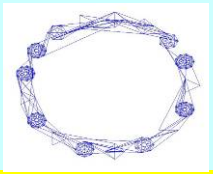
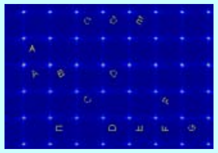
E.g., understand multi-programmer development workflow

# Type of Testbeds

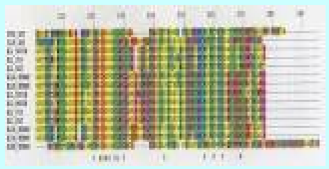
We are experimenting with a series of testbeds ranging in size

**Classroom assignments**

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

**Compact applications**

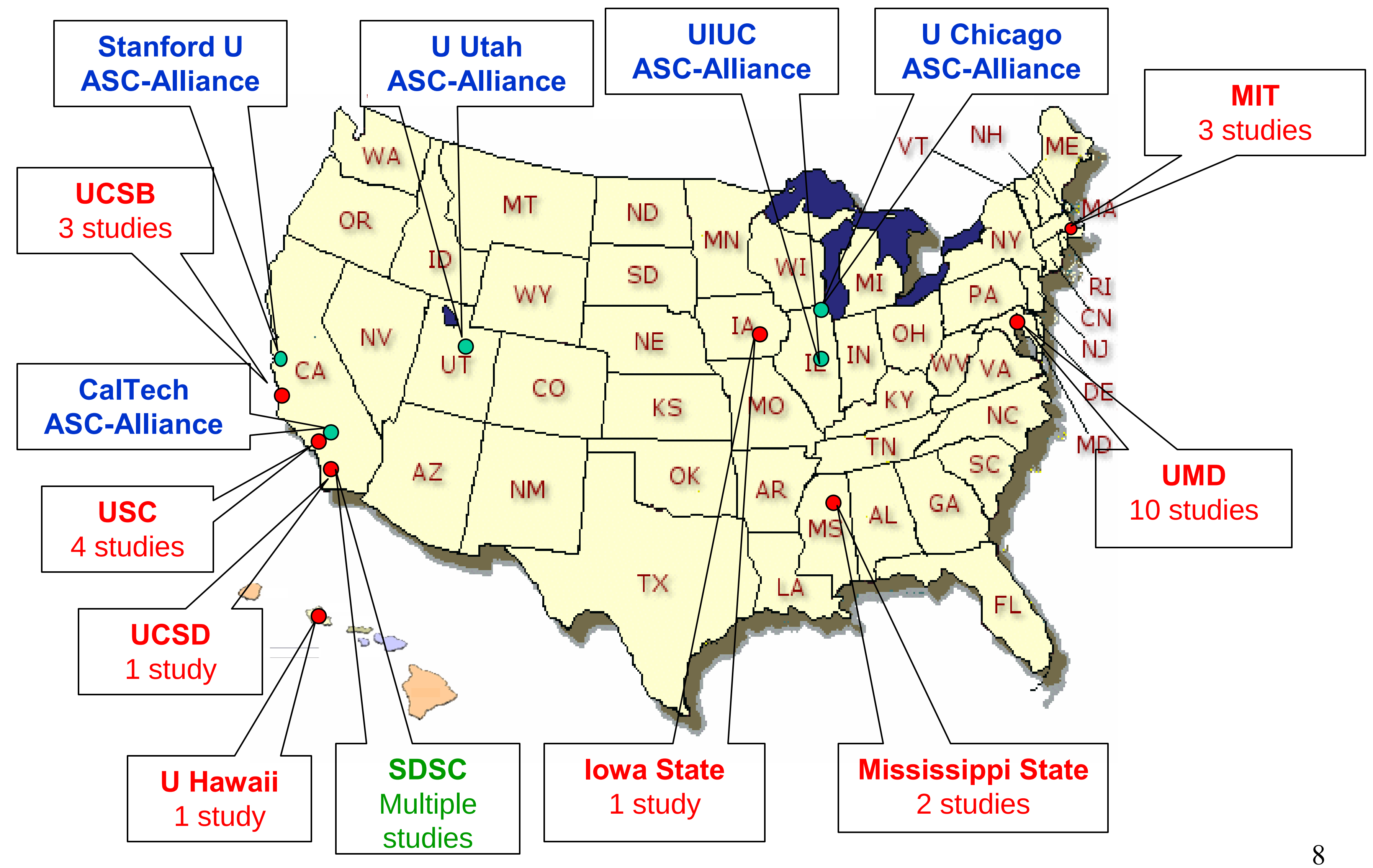


Bioinformatics, graph theory, sensor & I/O: combination of kernels, e.g., Embarrassingly Parallel, Coherence, Broadcast, Nearest Neighbor, Reduction
Developed by experts testing key benchmarks

**Full scientific applications**

climate modeling, protein folding, ….)
Developed at ASCI Centers at 5 universities
Run at the San Diego Supercomputer Center

# Study Locations



# Approach: Learning over time

- **Pilot classroom studies** on single programmer assignments
  - Identify variables, data collection problems, workflows, experimental designs
- Lead to **Observational Studies** on single programmers
  - Develop variables and data we can collect with confidence based upon our understanding of the problems
- Lead to **Controlled experiments** of single programmers
  - ction,

**Crawl before you walk before you run**

- Lead to **team projects** with graduate students
  - Study scale-up, multi-developer workflows,
- Lead to **professional developer studies**
  - Study scale-up, multi-developer workflows,
- Interviews with **developers and users** in a variety of environments…

## Approach: Learning over time Variables to Models

- **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
## Folklore to Supported Hypotheses

- **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., OpenMP offers more speedup for novices in a shorter amount of time when the problem is more computationally-based than communication based.

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

# Classroom Studies
# Models & Problems

| | MPI | OpenMP | UPC/CAF | Matlab*P | XMT-C |
|---|---|---|---|---|---|
| **Embarrassingly parallel** | | | | | |
| Buffon-Laplace needle problem | 2 | 2 | | 2 | |
| Dense matrix-vector multiply | 1 | 1 | | | |
| **Nearest neighbor** | | | | | |
| Game of life | 3 | 1 | 1 | 1 | |
| Sharks & fishes | 2 | 2 | 1 | | |
| Grid of resistors | 1 | 1 | | 1 | |
| Laplace's equation | 1 | | | 1 | |
| Quantum dynamics | 1 | 1 | | | |
| **All-to-all** | | | | | |
| Sparse matrix-vector multiply | 1 | | | | 1 |
| Sparse conjugate gradient | 2 | 2 | 1 | 1 | |
| Matrix power via prefix | 1 | 1 | | | |
| **Other** | | | | | |
| Sorting | 2 | 1 | | | |
| **(Shared memory)** | | | | | |
| LU decomposition | | 1 | | | |
| Shallow water model | | 1 | | | |
| Randomized selection | | | | | 2 |
| Breadth-first search | | | | | 1 |

# Models & Problems

| Problem | serial | MPI | OpenMP | Matlab*P | XMT-C | Co-Array Fortran | UPC | Hybrid MPI-OpenMP |
|---|---|---|---|---|---|---|---|---|
| game of life | A-F03 B-S04 E-S04 E-S05 | A-F03 B-S04 E-S04 E-S05 E-S06 | E-S04 E-S05 | E-S05 | | E-S05 E-S06 | E-S05 E-S06 | |
| SWIM | | | A-F03 | | | | | |
| Buffon-Laplace | D-S04 E-S04 | D-S04 E-S04 D-S06 | D-S04 E-S04 | D-S04 E-S04 D-S06 | | | | |
| Laplace's equation | D-S04 | D-S04 | D-S04 | D-S04 | | | | |
| sharks & fishes | G-F04 | G-F04 B-F05 | G-F04 B-F05 | | | G-F04 | | |
| grid of resistors | D-S04 | D-S04 | D-S04 | D-S04 | | | | |
| matrix power via prefix | | D-S05 D-S06 E-S06 | D-S05 | | | E-S06 | E-S06 | |
| sparse conjugate-gradient | | E-S05 E-S06 | | | | E-S06 | E-S06 | |
| dense matrix-vector multiply | G-F04 | G-F04 | G-F04 | | | | | |
| sparse matrix-vector multiply | C-S06 | E-S04 | | | C-S05 C-S06 | | | |
| sorting | E-S04 C-S06 | E-S04 I-S05 I-S06 | I-S05 | | C-S06 C-S06 | | | |
| quantum dynamics | | H-F04 H-F05 | | | | | | |
| molecular dynamics | | | | | | | | H-F05 |
| randomized selection | | | | | C-S04 | | | |
| breadth-first search | | | | | C-S06 | | | |
| LU decomposition | | | F-S04 | | | F-S06 | | |
| shortest path | | | F-S06 | | | | | |
| search for intelligent puzzles | | I-S06 | | | | | | |

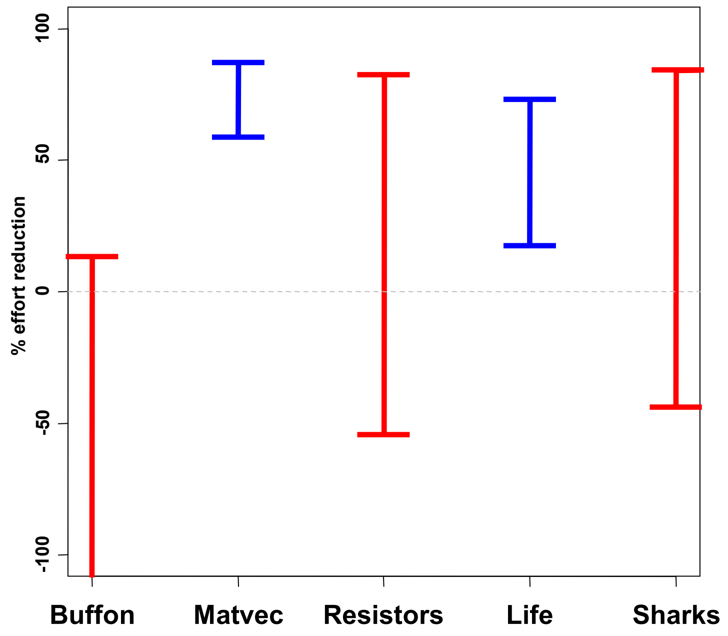
## Development Time Studies: Sample Results

Effect of model & problem on:

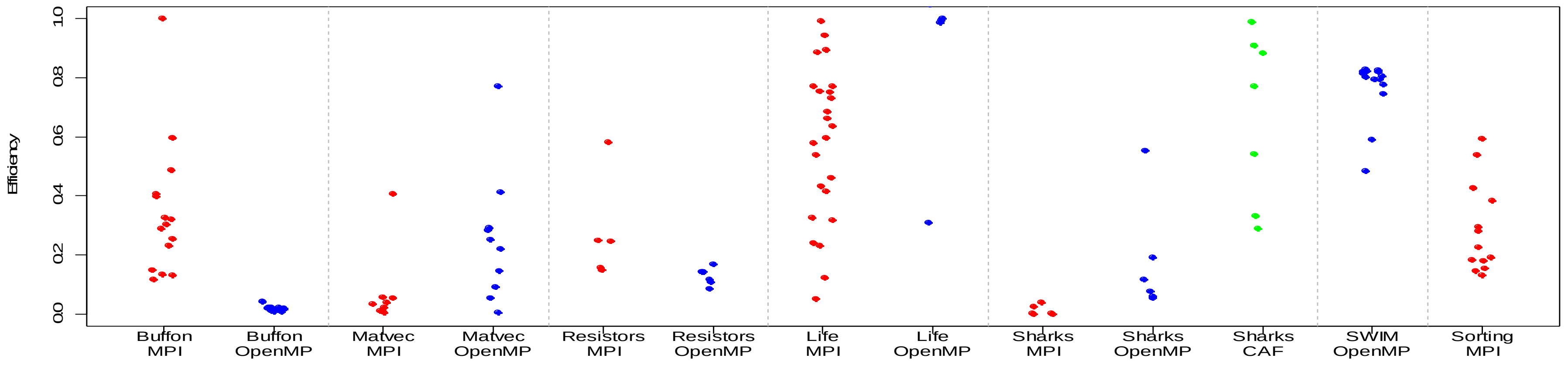
Effort

Performance

% Effort saved using OpenMP instead of MPI

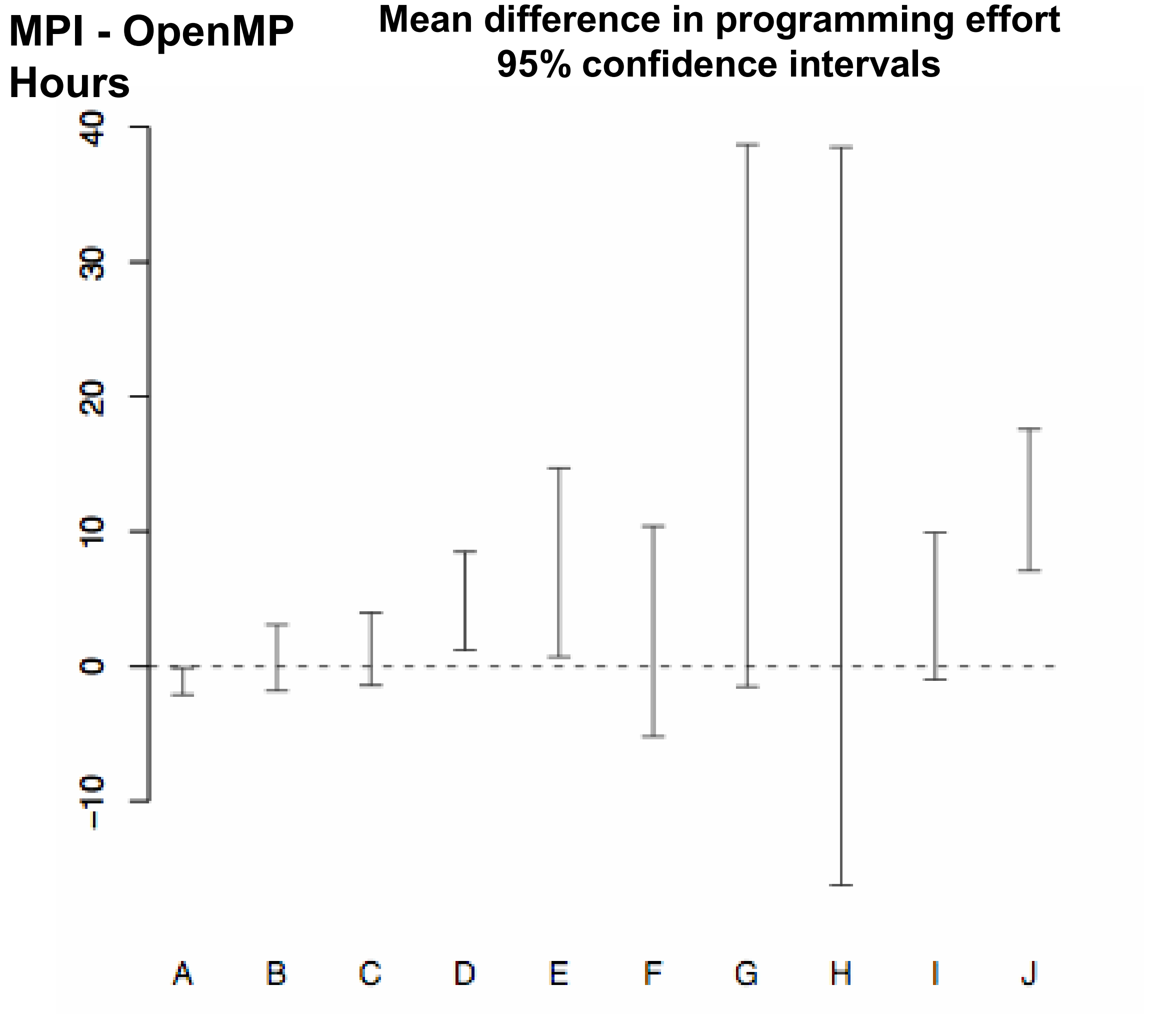


Performance (efficiency) by problem/language



# Development Time Studies: Comparing MPI & OpenMP

MPI vs. OpenMP
Mean difference in programming effort
95% confidence intervals



## Some Classroom Study Results

- **Threats to validity**:
  - Internal: measurement, problem experience
  - External: size, experience, motivation, new vs. existing, other development activities (e.g. porting), …
- **Acknowledging these threats, it appears that:**
  - 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

## ASC-Alliance Studies

- Extensive reuse of libraries, but no reuse of frameworks
  - Everyone has to write MPI code
- Codes are multi-language and run on remote machines
  - Many software tools won’t work in this environment
- 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!)
- 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!

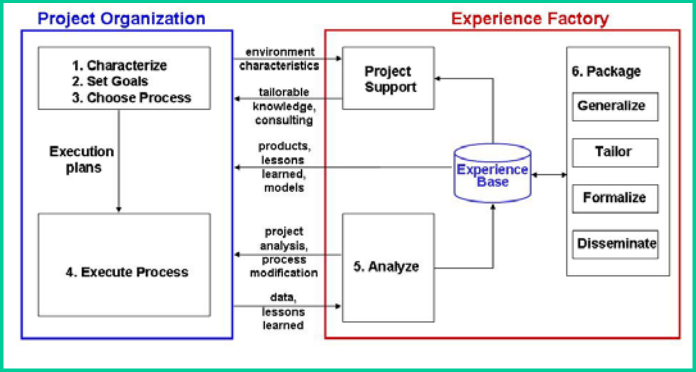
## SDSC Studies

- HPC 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
    - just learning parallel programming
- Queue is a major obstacle to productivity
- Performance is treated as a **constraint**, not a **goal to be maximized**
  - Performance is important until it is "good enough" for their machine allocation
- 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
- Many projects do not have anyone with a computer science background
- Visualization is regularly used for validation

## Productivity Research (w/ SDSC)

- SDSC Team
  - Allan Snavely, Nicole Wolter, Michael McCracken
- Do HPC users all have similar concerns and difficulties with productivity?
- Are users with the largest allocations and most experience the most productive?
- Is time to solution the limiting factor for productivity on HPC systems?
- Lack of publicity is the main roadblock to adoption of tools?
- Would HPC programmers demand dramatic performance improvements to consider major structural changes to their code?
- A computer science background is crucial to success in performance optimization?
- Is visualization not on the critical path to productivity in HPC?

# Where we need to go?



Software Engineering

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

## Study Team

**UMD**: Vic Basili, Jeff Hollingsworth, Marv Zelkowitz, Taiga Nakamura, Sima Asgari, Forrest Shull, Nico Zazworka, Rola Alameh, Daniela Suares Cruces
**UNL**: Lorin Hochstein
**MSU**: Jeff Carver
**UH**: Philip Johnson

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