Why Compilers Have Failed To Support HPC Programmers and What Can We Do About It

Saman Amarasinghe

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory

Impact of Languages and Compilers

- Languages and Compilers have drastically improved the programmer productivity
  - Ease of expression and construction of large programs
    - High Level Languages
    - Object Oriented Languages
  - Elimination of many classes of bugs
    - Managed Memory
    - Type Safety
  - Fully portable across all hardware
    - Instruction Level Parallelism
- …except in high performance programming!

Impact of Languages and Compilers in parallelism

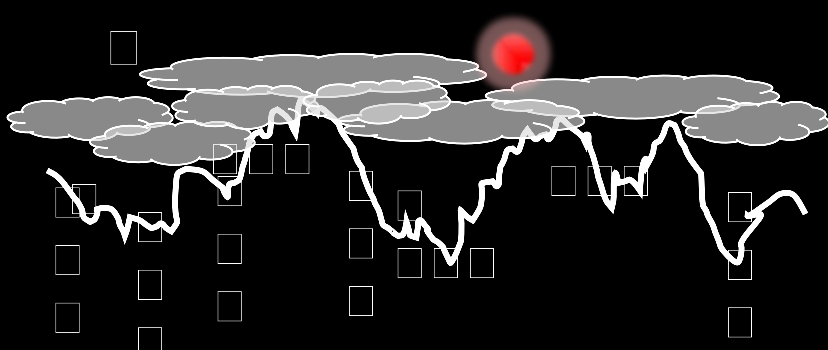
- Parallel programming still feels like assembly level programming
  - All the hardware features are fully exposed
    - Need to explicitly manage → no portability
  - Many classes of nasty bugs
    - Deadlocks, race conditions etc.

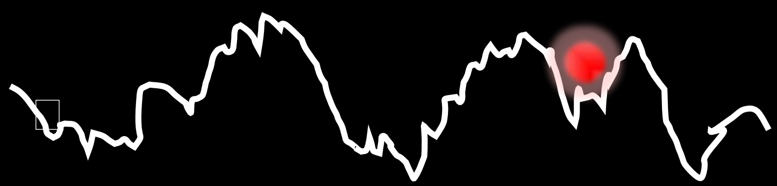
Success Criteria for a Compiler

1. Effective
2. Stable
3. Portable
4. Scalable
5. Simple

1: Effective

- Compiler optimizations has to select the best choice among all possibilities, but…
  - Options are obscured
    - Impossible to identify, evaluate, select
  - Options not available
    - In a local minima
    - Heroic effort needed to get out





- To be effective compiler
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user
  - Expose ones that are automatable and variable → hide from the user

2: Stable

- Simple change in the program should not drastically change the performance!
  - Otherwise need to understand the compiler inside-out
  - Programmers want to treat the compiler as a black box

3: Portable

- Work on the spectrum of current architectures
  - Terrascale, petascale
- Need to be “Future-Proof”
  - Ex: heterogeneous architectures
- Cannot hardcode parameters that’ll change

4: Scalable

- Works well on your small cluster is good
- …but will it work the same work on Jaguar?
- How about the exascale machines?

5: Simple

- Aggressive analysis and complex transformation lead to:
  - Buggy compilers!
    - Programmers want to trust their compiler!
    - How do you manage a software project when the compiler is broken?
  - Long time to develop
- Simple compiler $\Rightarrow$ fast compile-times
- Current compilers are too complex!

Compiler	Lines of Code
GNU GCC	~ 1.2 million
SUIF	~ 250,000
Open Research Compiler	~3.5 million
Trimaran	~ 800,000
StreamIt	~ 300,000

A Success Story: Register Allocation

- Effective
  - Every architecture has registers at the bottom of the memory hierarchy
  - All the registers were hidden from the users
    - Early C let the users bound registers to variables, but now hidden from the user
  - Users are exposed to identifying reg allocatable variables (i.e. with volatile)
  - Allocating a variable to a register reduce mem bandwidth → clear winner
- Stable
  - Local optimization. If you miss one, no global consequence
- Portable
  - Variations between hardware (# of regs, special purpose regs) is exposed and managed by the compiler
- Scalable
  - Local problem, out of Moore's curve → scaling is not an issue
- Simple
  - Graph coloring and spilling heuristics is (now) trivial

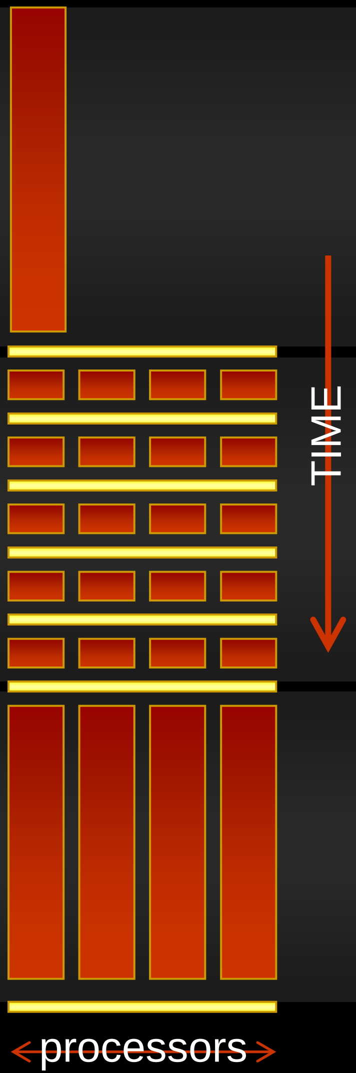
The Dream: Automatic Parallelization

- Identify loops where each iteration can run in parallel
  - DOALL parallelism
- What Matters
  - Parallelism Coverage
  - Parallelism Granularity

```
TDT = DT
 MP1 = M+1
 NP1 = N+1
 EL = N*DX
 PI = 4.D0*ATAN(1.D0)
 TPI = PI+PI
 DI = TPI/M
 DJ = TPI/N
 PCF = PI*PI*A*A/(EL*EL)
```

```
 DO 50 J=1,NP1
    DO 50 I=1,MP1
         PSI(I,J) = A*SIN((
         I-.5D0)*DI)*
         SIN((J-.5D0)*DJ)
         P(I,J) = PCF*(COS(2.D0)
CONTINUE
```

```
 DO 60 J=1,N
      DO 60 I=1,M
         U(I+1,J) = -(PSI(I+1,J+1)
         -PSI(I+1,J))/DY
         V(I,J+1) = (PSI(I+1,J+1)-
         PSI(I,J+1))/DX
 CONTINUE
```



Why Automatic Parallelism Failed

- Lack of Effectiveness
  - Sequential description obscures inherent parallelism
  - Need heroic analysis
- Lack of Scalability
  - Amdhal's law: increased parallelism → more parallelism coverage
  - Need more heroic analysis
- Lack of Stability
  - Granularity of Parallelism
  - Small changes have a large impact
    - Parallelize one additional statement → change the granularity
  - Needs even more heroic analysis
- Lack of Simplicity
  - All these heroic analyses → A hugely complex compiler

The Reality: MPI + X

- All the burden on the programmer
  - Parallelization
  - Computation and Data partitioning
  - Communication orchestration

Why Compilers will not succeed with MPI+X

- Lack of Effectiveness
  - Programmer binds most important decisions
  - Not too much choice exposed to the compiler
- Lack of Portability
  - Data partitioning and communication orchestration
    - Early binding to the given architecture
    - Heroic analysis will be needed to change automatically
  - MP+OpenMP+Cuda+???
    - The partitioning match the current components
    - Heterogeneous mix will change in the future
- Lack of Scalability
  - Hard to scale when hard bound to current machines

If we have a Revolution, what should it be?

- A new programming model/language that….
  - Will take much of the burden of away from the programmer
    - Managing the architectural features
    - Tuning for performance
  - Will make some classes of hard problems completely go away
    - No race conditions or deadlocks
  - Will make is possible for the compiler to “do the right thing”
    - Able to optimize by taking advantage of all the capabilities
    - Able to provide performance portability for current and future machines
  - Will make is possible for experts to “help” the compiler
    - A performance guru can provide patterns and transformations that are specific to the given application
- A new compiler that will not let the programmers down!

Selecting between the programmer and the compiler

- Let the programmer handle features that are impossible to automate
  But…make them constant across all current and future architectures
  - Get the programmers to expose maximum concurrency inherent to the algorithm
  - Get the programmers to over partition the data (perhaps hierarchically)
  - Get the programmer to provide more than one choice of algorithm and data partition

Selecting between the programmer and the compiler

- Let the compiler handle features that change across architectures
  - Managing parallelism
  - Managing heterogeneity
  - Managing data partitioning
  - Managing communication orchestration

What happens if these are still too hard for the compiler to handle?

- Provide hooks so expert performance gurus can intervene when needed
- Invest in developing compiler technology
- Wait patiently until the compiler people get it(hopefully!) working

Problem with High Performance Languages

- There are no new ideas in high performance languages
  - No new constructs
  - No new programming models

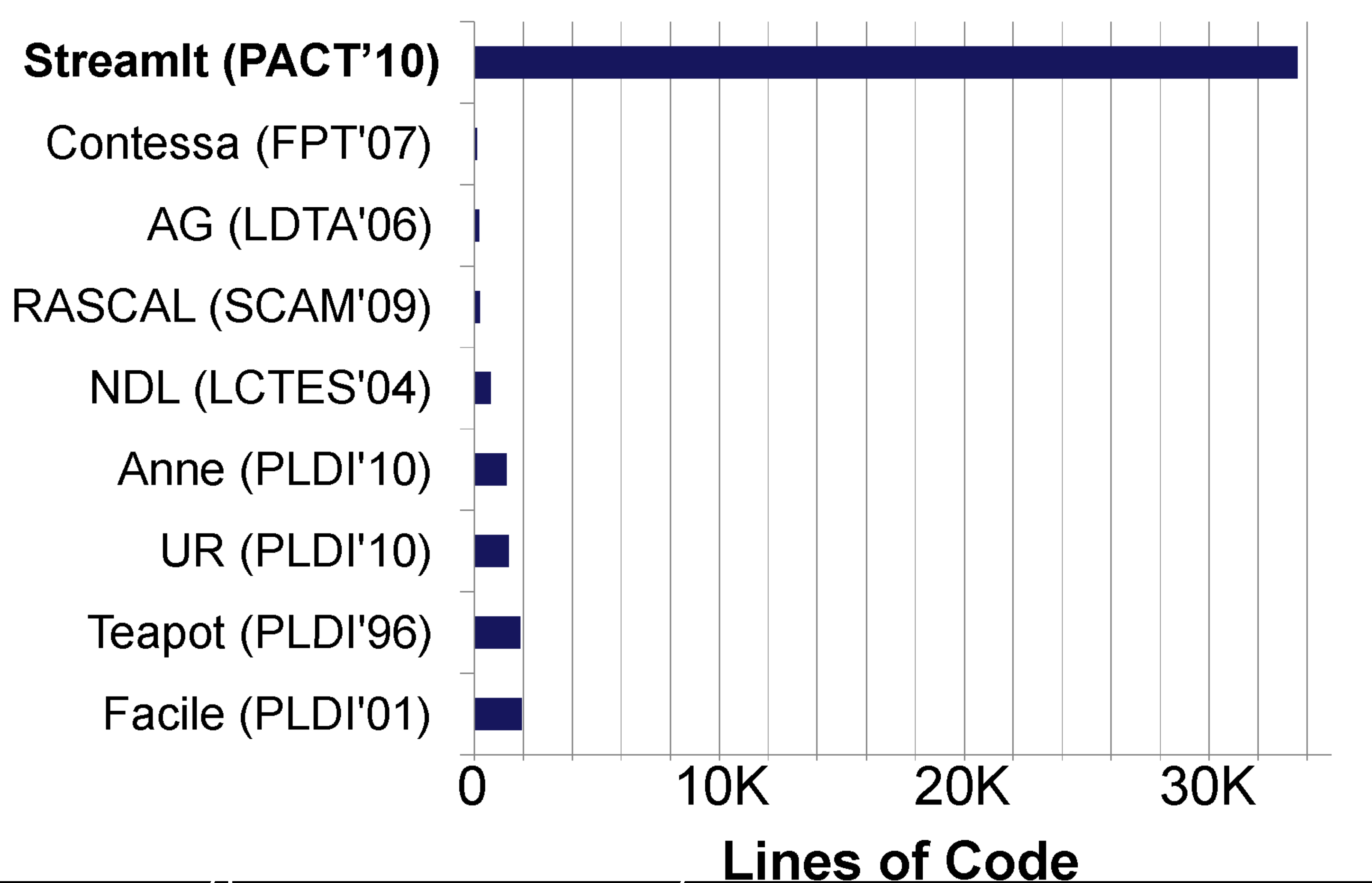
- Either…
  - We have discovered all there is to find
  - We have lost the capability to find new ones

Why it is hard to evolve a new language (feature)

- Test languages are different from production languages
  - Test language: experiment with a couple of features
  - Production language: feature complete
    - Integrate good features from multiple (test) languages
- Languages need to evolve
  - Hard to get it right the first time
  - Most user interface designs processes are set around rapid evolution with ample user feedback
- Need input from programmers to evolve
  - Need a lot of programmers to use the language
    - Different programmers think differently
  - Need programmers to use it for a long time
    - First impression is not what makes a good language
    - Measure the productivity of a trained programmer in the language

Why it is hard to evolve a new language (feature)

- Market forces work against new languages
  - Primary criteria for adoption is large number of existing users
- There is nothing in it for a programmer
  - Hard to make a long-term investment
    - The language may not last
    - At best, it'll keep changing
  - Has to deal with bugs
    - The compiler will be buggy
  - Has to deal with incomplete systems
    - Important features will be missing
    - Tools will be missing
  - More promise than reality
    - Compiler optimizable does not mean optimizations will be implemented…or works well.


StreamIt (PACT'10)
Contessa (FPT'07)
AG (LDTA'06)
RASCAL (SCAM'09)
NDL (LCTES'04)
Anne (PLDI'10)
UR (PLDI'10)
Teapot (PLDI'96)
Facile (PLDI'01)
0
10K
20K
30K
Lines of Code

Proposal: A National Center for Programming Language Evaluation

- A Virtual Center
  - Access to many professional programmers with difference skills
  - Infrastructure for scientific and unbiased evaluation
- Evaluation process akin to Drug Trials
  - Stage 1:
    - Select 20 language/feature projects
    - One week evaluation with 5 to 15 programmers
    - Write a set of small kernels
  - Stage 2:
    - Down select 4 to 5 projects
    - 3 to 6 month evaluation by 20 to 40 programmers
    - In one or two teams, develop a substantial application
  - Stage 3:
    - Down select 1 to 2 projects
    - Provide support to build/improve the tools and the compiler
    - One year effort by 50 to 100 programmers to port a real system