UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

Languages and compilers for parallel computing Where do we go from here? Research directions at mid-century

David Padua

illinois.edu

The problem

- At some point it seemed like the promise of parallelism was never going to happen
- But finally parallelism is now ubiquitous in hardware like never before
  - Vector extensions
  - Instruction level parallelism
  - Multicores
  - Clusters
- In time, more applications must take advantage of parallelism
  - Parallelism will be main source of improvement in speed and power consumption.
- Parallel programming must become the norm.
  - But we are far from it.

The challenge

- Make parallel programming the norm.
- We would like to build on the achievements of ~50 years of research.
- Specific challenges:
  - **Notations:** Parallel programming introduces additional complexities. A dip in productivity is expected. We need to reduce its magnitude by providing easy to use / accept notations.
  - **Portability:** Want performance portable programs across classes of parallel machines.
  - **Optimization:** Parallel programming introduces new challenges when programming for performance. And hardware parallelism is mainly about performance.
  - **Correctness:** New classes of defects
    - Unwanted non-determinacy
    - Deadlocks
- The pages of LCPC proceedings a filled with proposals along these lines, but the problem is fairly much unresolved.

Strategies to address the challenges

- Address the Notation problem:
  - Want parallelism to become standard in all popular forms of programming – not the case today.
    - CS courses use sequential notation
    - Programmers often ignore performance/scalability issues
    - Standards are beginning to evolve, but this seems to have little effect.
- Address Automation
  - Optimization for performance.
    - Autotuners
    - Compilers
    - Not widely used to attain parallelism. Lack of effectiveness
  - Defect detection and avoidance
    - Some tools/static analysis ideas (e.g. DPJ)
    - Not clear how useful

Whither are we bound ?

- Some say that the problem cannot be solved in a radical form and are content with today's solutions
  - Nothing better than MPI
  - Automation is impossible. Paraphrasing : "Grad students: Don't work on that problem it will destroy your careers!" .
- What is our position?
  - We don't want to sound too much as visionaries. The failures of the past hang on our backs.
    - HPF, Autovectorization, UPC?,...

PARALLEL PROGRAMMING NOTATIONS

1. Dynamic languages: No real parallelism
2. Extensions for conventional languages:
   - Hierarchically Tiled Arrays
   - Abstractions for stencil computations
3. Very high-level notation:
   - Hydra

Dynamic Languages

- Dynamic languages are for
  - Productivity. They “Make programmers super productive”.
  - Not performance
- DLs are typically slow.
  - 10-100 (1000 ?) times slower than corresponding C or Fortran
- Sufficiently fast for many problems and excellent for prototyping in all cases
  - But must manually rewrite prototype if performance is needed.

Parallel Programming with Dynamic Languages

- Not always accepted by the DL community
  - Hearsay: javascript designers are unwilling to add parallel extensions.
  - Some in the python community prefer not to remove GIL – serial computing simplifies matters.
  - In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.
- Not (always) great for performance
  - Not much of an effort is made for a highly efficient, effective form of parallelism.
    - For example, Python's GIL and its implementation.
    - In MATLAB, programmer controlled communication from desktop to worker.

- Not (always) great to facilitate expressing parallelism (productivity)
  - In some cases (e.g. MATLAB) parallel programing constructs were not part of the language at the beginning.
  - Sharing of data not always possible.
    - Python it seems that arrays can be shared between processes, but not other classes of data.
    - In MATLAB, there is no shared memory.
  - Message passing is the preferred form of communication.
    - Process to process in the case of Python.
    - Client to worker in the case of MATLAB
    - MATLAB's parfor has complex design rules

- There are reasons to improve the current situation
- Parallelism might be necessary for dynamic languages to have a future in the multicore era.
  - Lack of parallelism would mean no performance improvement across machine generations.
  - DLs are not totally performance oblivious. They are enabled by very powerful machines.
- When parallelism is explicit
  - For some problems it helps productivity
  - Enable prototyping of high-performing parallel codes.
  - Super productive parallel programming ?
- Can parallelism be used to close the performance gap with conventional languages ?

Whither are we bound ?

- Need to address the problem of performance.
- Difficult and unresolved for general languages like Python.
- Easier for MATLAB/R, but not implemented either. Why ?

Extensions for parallelism

- Many notations developed.
- None widely used for programming

Accomplishments of the last decades in programming notation

- Much has been accomplished
- Widely used parallel programming notations
  - Distributed memory (SPMD/MPI) and
  - Shared memory (pthreads/OpenMP/TBB/ Cilk/ArBB).

Languages

- OpenMP constitutes an important advance, but its most important contribution was to unify the syntax of the 1980s (Cray, Sequent, Alliant, Convex, IBM,…).
- MPI has been extraordinarily effective.
- Both have mainly been used for numerical computing. Both are widely considered as “low level”.

The future

- Higher level notations
- We may want to avoid changes in programming style.
  - Sequential programming style?
- The solution is to use abstractions.

Array operations in MATLAB

- An example of abstractions are array operations.
- They are not only appropriate for parallelism, but also to better represent computations.
- In fact, the first uses of array operations does not seem to be related to parallelism. E.g. Iverson's APL (ca. 1960). Array operations are also powerful higher level abstractions for sequential computing
- Today, MATLAB is a good example of language extensions for vector operations

Array operations in MATLAB

Matrix addition in scalar mode

```
for i=1:m,
    for j=1:1,
    c(i,j)= a(i,j) + b(i,j);
    end
end
```

Matrix addition in array notation

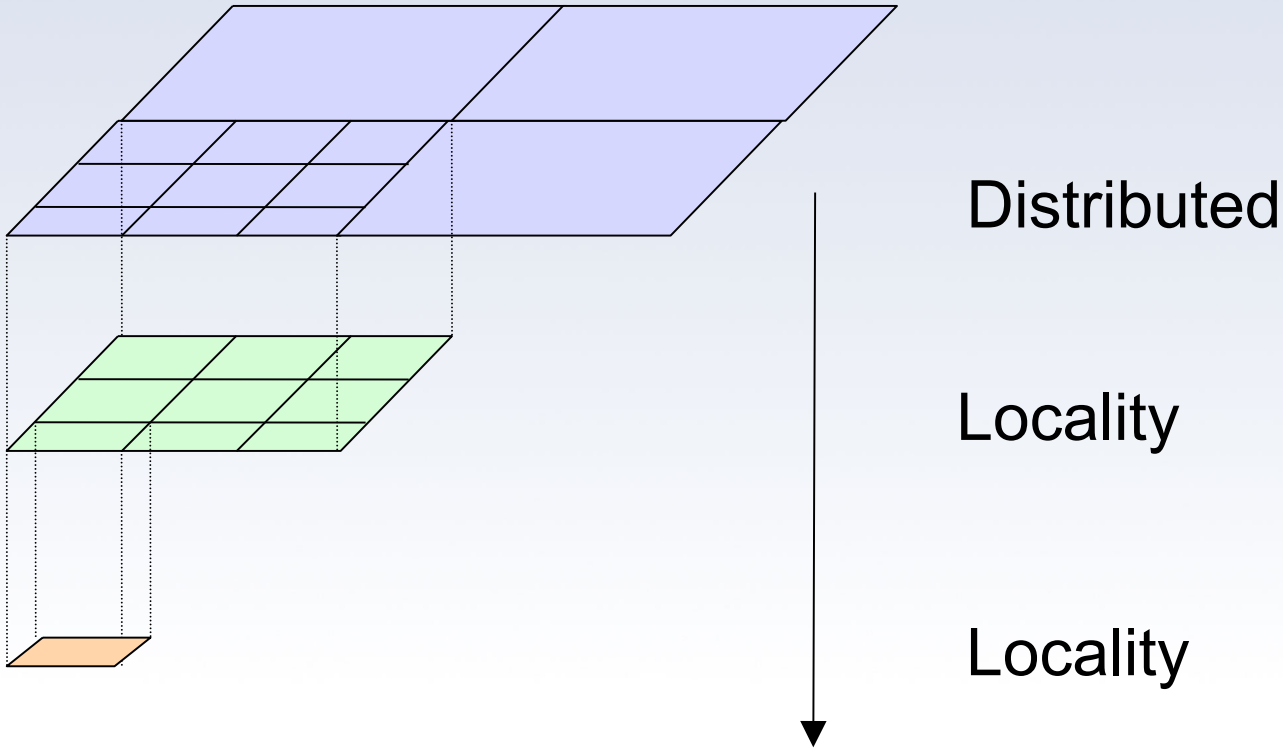
```
c = a + b;
```

An important generalization: Hierarchically Tiled Arrays

Ganesh Bikshandi, James Brodman, Basilio Fraguela, Maria Garzaran, Jia Guo, Christoph von Praun, David Padua

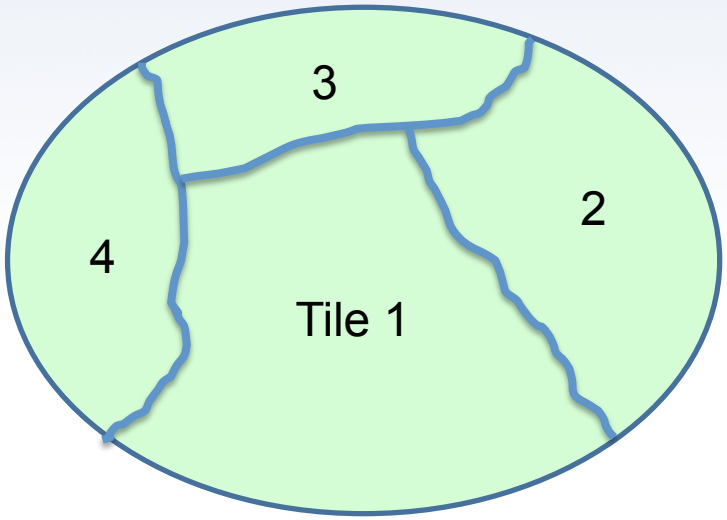
- Recognizes the importance of blocking/ tiling for locality and parallel programming.
- Makes tiles first class objects.
  - Referenced explicitly.
  - Manipulated using array operations such as reductions, gather, etc..

A hierarchically tiled array



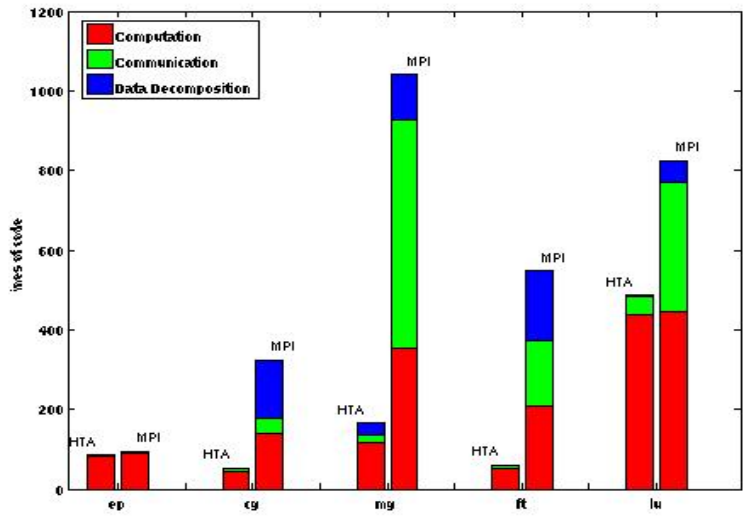
Tiled Sets

- The mapping function places the data in a set into tiles
- Trivial for Arrays
- More difficult for Irregular data structures



Compact code

- Overall, code quality much better than that of conventional notations



- James C. Brodman, G. Carl Evans, Murat Manguoglu, Ahmed H. Sameh, María Jesús Garzarán, David A. Padua: A Parallel Numerical Solver Using Hierarchically Tiled Arrays. LCPC 2010: 46-61

Ongoing and future work

- Developing graph algorithms for distributed memory machines. Next step: identify API for this type of problems.
- Also, looking into linear algebra solutions to graph algorithms. Reordering.
- Study of effectiveness of high-level notations for portability, exascale system

Whither are we bound ?

- Need to popularize this approach.
- So far, many failures in the parallel realm
  - HPF
  - ArBB
- But successes in the conventional world
  - MATLAB
  - R
- What is the future of Fortran array extensions/Cilk array extensions?

Very high level notation

- The perennial promise:
  - Very high level notation
  - Domain-specific notations
- Some progress: e.g. SPIRAL
  - Still a promise

Hydra

Denis Barthou, Alexandre Duchateau

- Compile linear algebra equations
  - Compute X for $L * X - X * U = C$ [CTSY]
  - Compute L and U for $L * U = A$ [LU]
- Generate efficient task parallel code
  - Identify tasks
  - Generate task dependence graph

Description Language - Operands

```
%% Operands
X: Unknown Matrix
L: Lower Triangular Square Matrix
U: Upper Triangular Square Matrix
C: Square Matrix

%% Equation
L*X-X*U=C
```

- All operands
- (Type inference)

- Status
  - Known, Unknown
- Shape
  - Triangular, diagonal
- Type
  - Matrix, (vector, scalar)
- Modifiers (transpose)
- (Sizes)
- (Density)

Description language - Equation

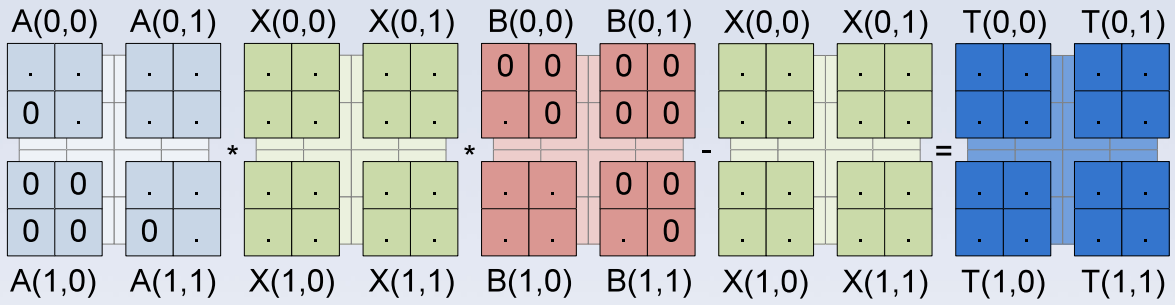
```
%% Operands
X: Unknown Matrix
L: Lower Triangular Square Matrix
U: Upper Triangular Square Matrix
C: Square Matrix

%% Equation
L*X-X*U=C
```

- Simple equations
  - Assignments
    - X = A*B
- Solvers
  - LU
  - Triangular Sylvester
  - L*X=B
  - Cholesky
- Base for decomposition

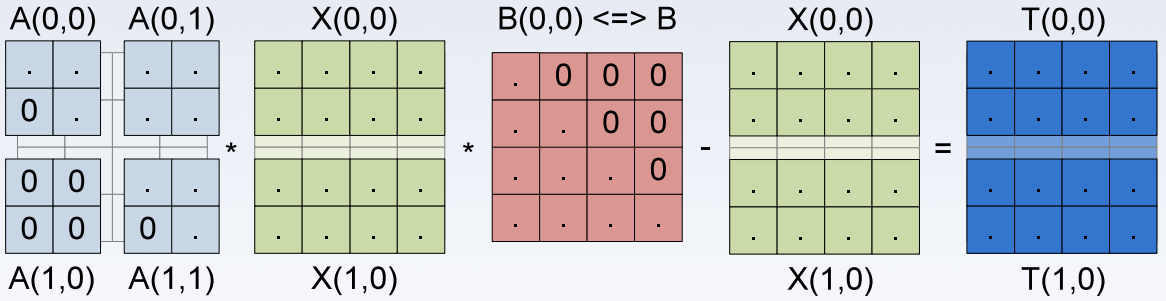
Valid Blockings – DTSY

A*X*B – X = C | A lower triangular, B upper triangular

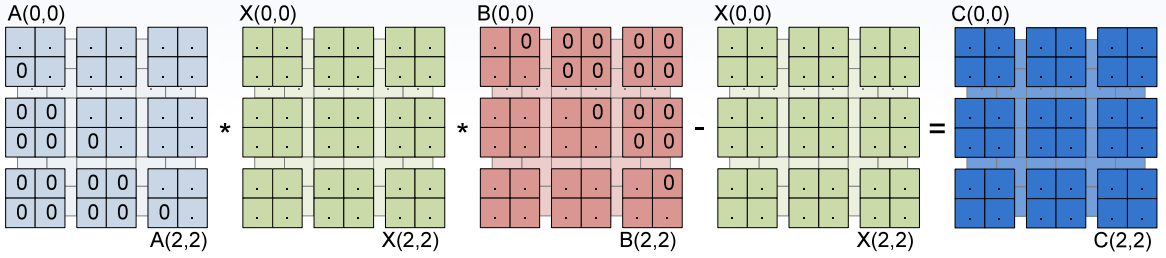
xA = yA = xX = 2
xB = yB = yX = 2



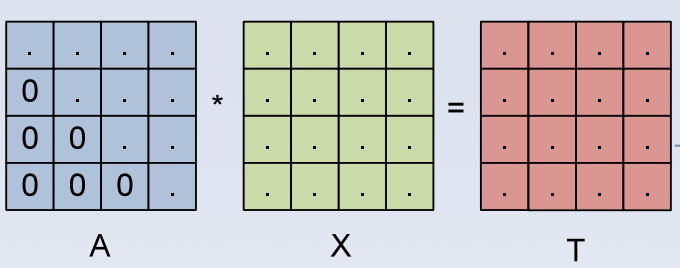
xA = yA = xX = 2
xB = yB = yX = 1



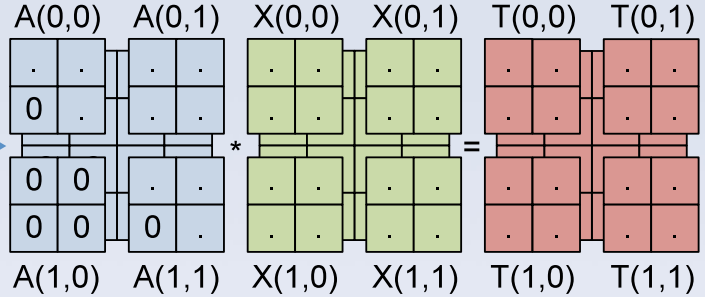
xA = yA = xX = 3
xB = yB = yX = 3



Derivation example



blocking



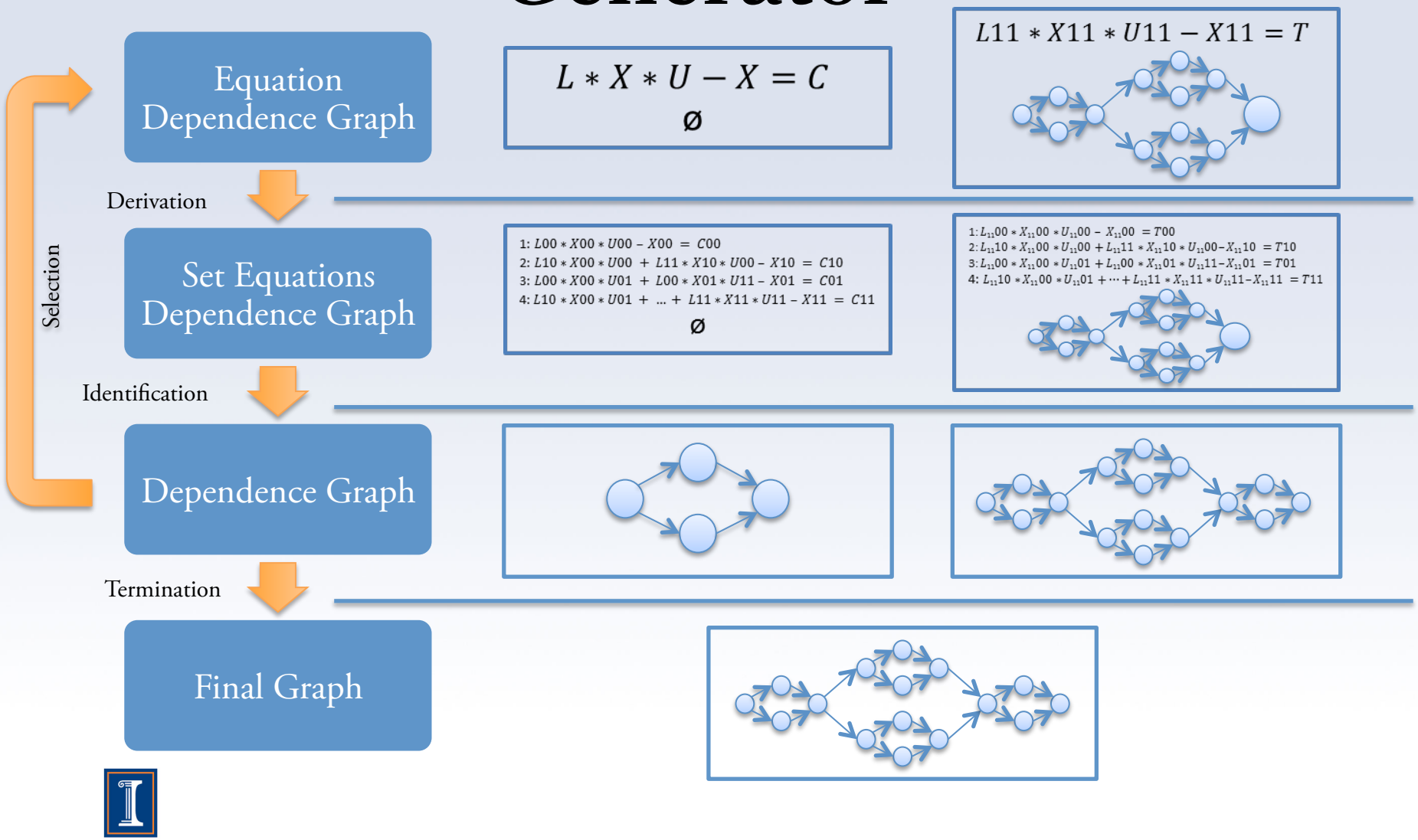
Sym. Exec.

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) = A(1,0)*X(0,0) + A(1,1)*X(1,0)
T(1,1) = A(1,0)*X(0,1) + A(1,1)*X(1,1)
```

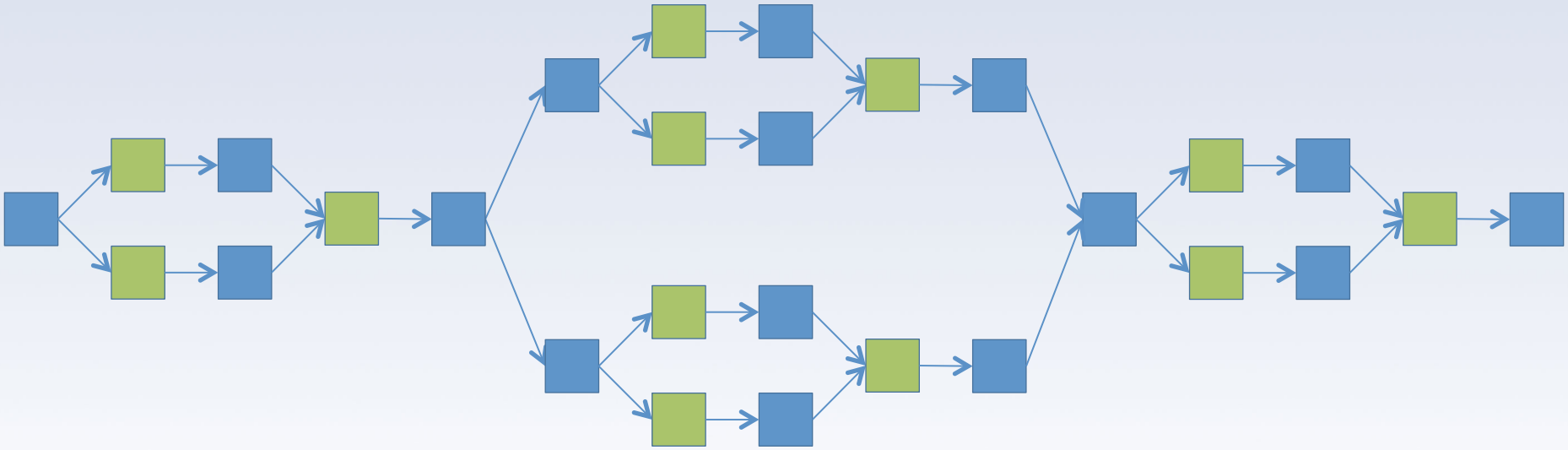
Removal of 0-computation

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) =                 A(1,1)*X(1,0)
T(1,1) =                 A(1,1)*X(1,1)
```

Generator


Equation Dependence Graph
$L * X * U - X = C$
Ø
$L11 * X11 * U11 - X11 = T$
Derivation
Selection
Set Equations Dependence Graph
1: $L00 * X00 * U00 - X00 = C00$
2: $L10 * X00 * U00 + L11 * X10 * U00 - X10 = C10$
3: $L00 * X00 * U01 + L00 * X01 * U11 - X01 = C01$
4: $L10 * X00 * U01 + \ldots + L11 * X11 * U11 - X11 = C11$
Ø
1: $L_{11}00 * X_{11}00 * U_{11}00 - X_{11}00 = T00$
2: $L_{11}10 * X_{11}00 * U_{11}00 + L_{11}11 * X_{11}10 * U_{11}00 - X_{11}10 = T10$
3: $L_{11}00 * X_{11}00 * U_{11}01 + L_{11}00 * X_{11}01 * U_{11}11 - X_{11}01 = T01$
4: $L_{11}10 * X_{11}00 * U_{11}01 + \cdots + L_{11}11 * X_{11}11 * U_{11}11 - X_{11}11 = T11$
Identification
Dependence Graph
Termination
Final Graph

Simple graph example



Whither are we bound ?

- Need to popularize these notations.
- Need to incorporate them in conventional languages.
- Robust implementations

COMPILER EVALUATION

Evaluation of vectorization

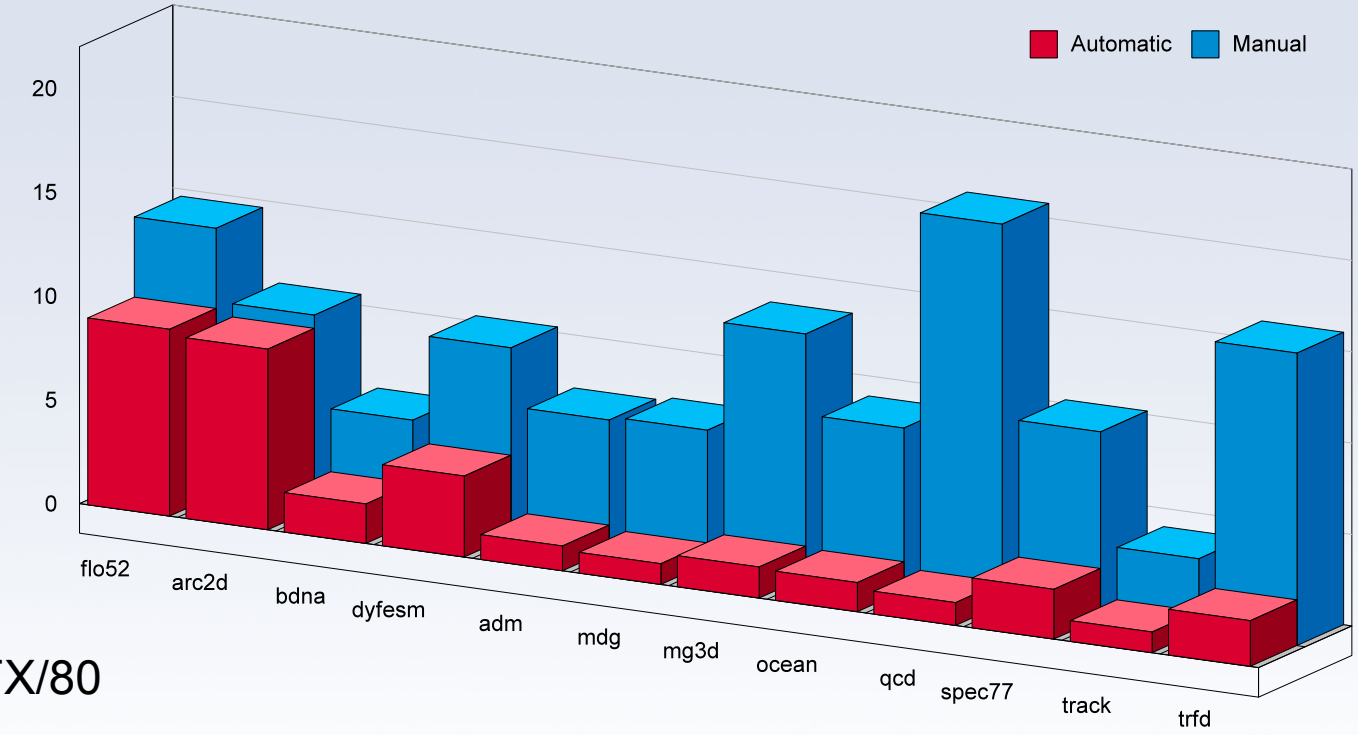
Purpose of compilers

- Bridge the gap between programmer's world and machine world. Between readable/easy to maintain code and unreadable high-performing code.
- The idiosyncrasies of multicore machines, however interesting in our eyes, are more a problem than a solution.
- In an ideal world, compilers or related tools should hide these idiosyncrasies.
- But, what is the hope of this happening today ?

How well do compilers work ?

- Evidence accumulated for many years show that compilers today do not meet their original goal.
- Problems at all levels:
  - Detection of parallelism
  - Vectorization
  - Locality enhancement
  - Traditional compilation
- I'll show only results from our research group.

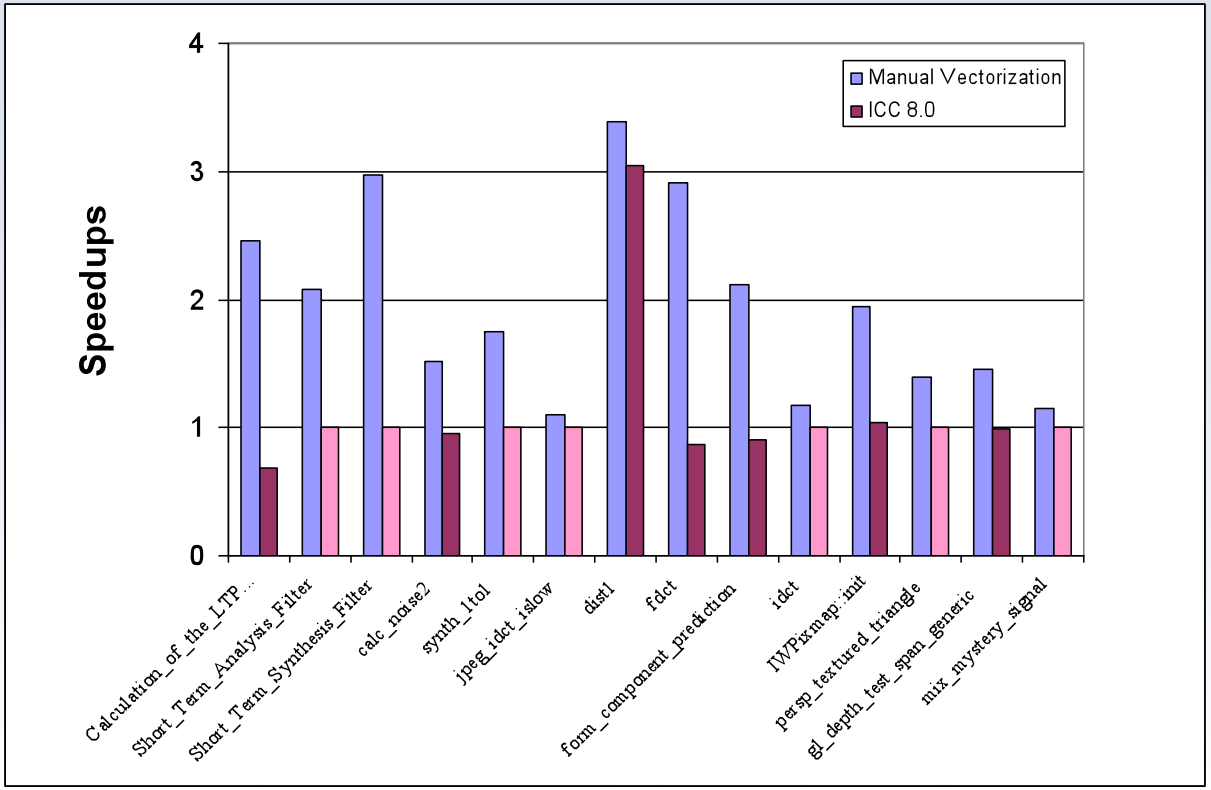
How well do they work ?
Automatic detection of parallelism



Alliant FX/80

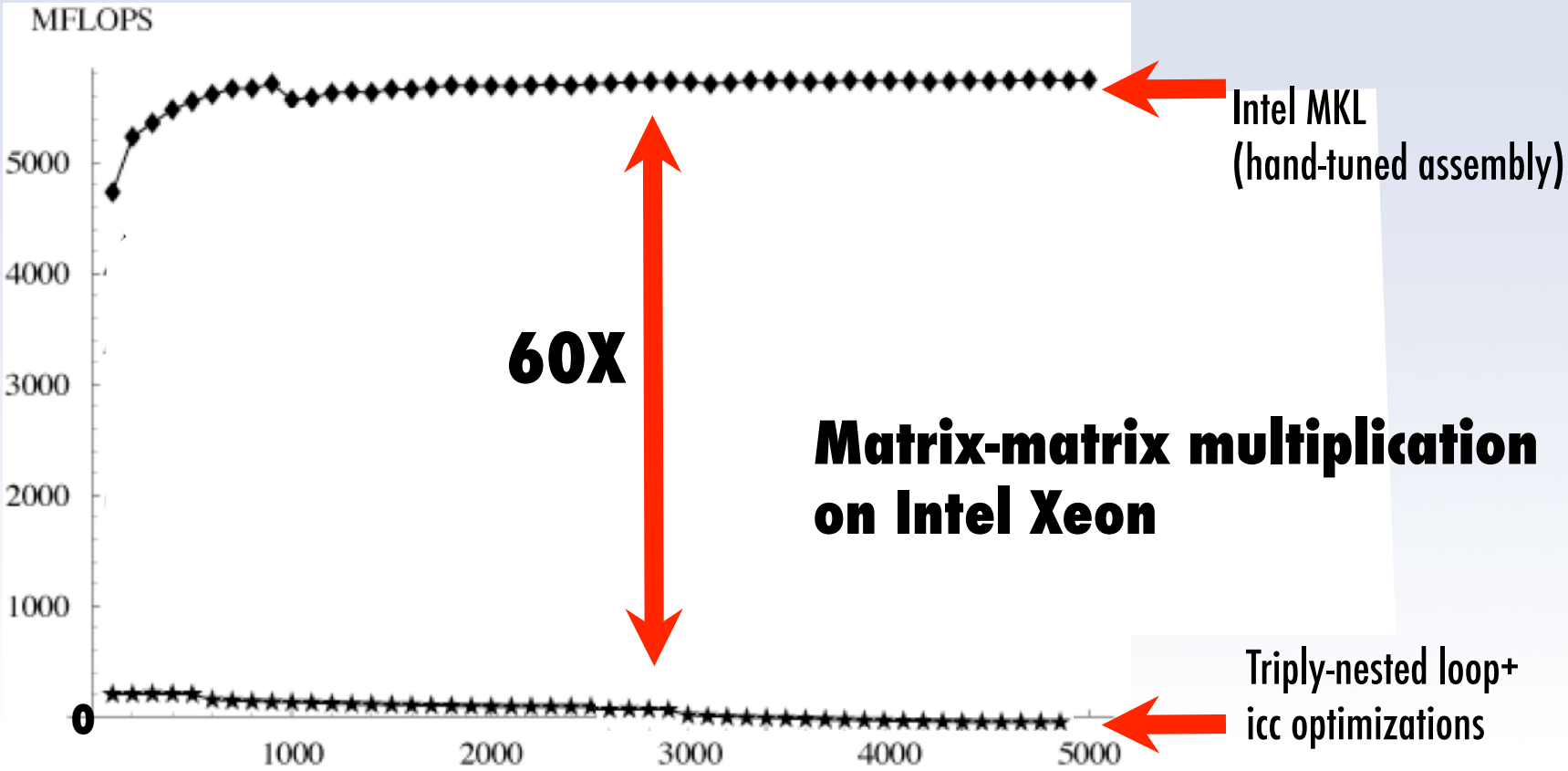
R. Eigenmann, J. Hoeflinger, D. Padua On the Automatic Parallelization of the Perfect Benchmarks. IEEE TPDS, Jan. 1998.

How well do they work ?
Vectorization



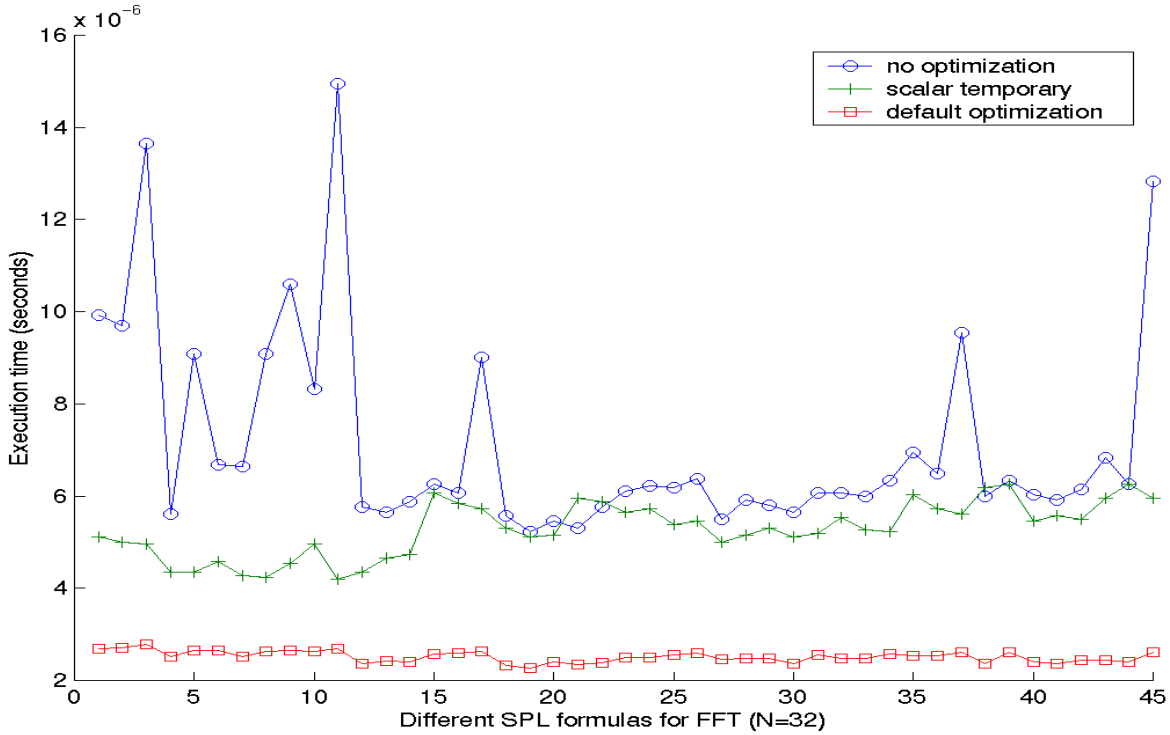
G. Ren, P. Wu, and D. Padua: An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. IPDPS 2005

How well do they work ?
Locality enhancement



K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill.
Is Search Really Necessary to Generate High-Performance BLAS?
Proceedings of the IEEE. February 2005.

How well do they work ?
Scalar optimizations



J. Xiong, J. Johnson, and D Padua. *SPL: A Language and Compiler for DSP Algorithms.* PLDI 2001

What to do ?

- We must understand better the effectiveness of today's compilers.
  - How far from the optimum ?
- One thing is certain: part of the problem is implementation. Compilers are of uneven quality. Need better compiler development tools.
- But there is also the need for better translation technology.

What to do ?

- One important issue that must be addressed is optimization strategy.
- For while we understand somewhat how to parse, analyze, and transform programs. The optimization process is poorly understood.
- A manifestation of this is that increasing the optimization level sometimes reduces performance. Another is the recent interest in search strategies for best compiler combination of compiler switches.

What to do ?

- The use of machine learning is an increasingly popular approach, but analytical models although more difficult have the great advantage that they rely on our rationality rather than throwing dice.

Obstacles

- Several factors conspire against progress in program optimization
  - The myth that the automatic optimization problem is solved or insurmountable.
  - The natural desire to work on fashionable problems and “low hanging fruits”

Evaluation of autovectorization

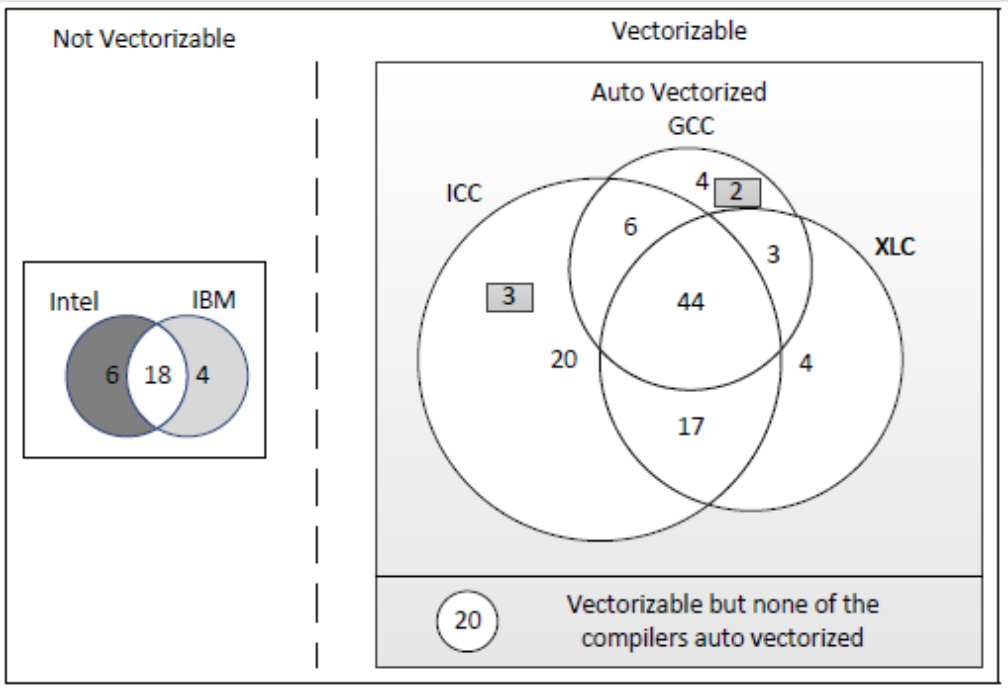
Saeed Maleki, Seth Abraham, Bob Kuhn, Maria J. Garzaran, and David Padua.

- Initiated as part of the Blue Waters Project.
- How effectively can compilers compile conventional C code onto microprocessors vector extensions ?

- Saeed Maleki, Yaoqing Gao, Maria J. Garzaran, Tommy Wong and David Padua. An Evaluation of Vectorizing Compilers. In Proc. of the International Conference on Parallel Architectures and Compilation Techniques, October 2011.

- Evaluation based on
  - Callahan, Dongarra, Levine loops (translated into C)
  - PACT (Petascale Application Collaboration Team) codes
    - DNS
    - MILC
  - Media Bench II
- Consider ICC, XLC, GCC

S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. An Evaluation of Vectorizing Compilers. PACT. 2011.


Not Vectorizable
Intel
IBM
6
18
4
Vectorizable
Auto Vectorized
GCC
ICC
XLC
4
2
6
3
3
44
20
4
17
20
Vectorizable but none of the compilers auto vectorized

Method	XLC	ICC
Vectorizable	124(82.12%)	127(84.11%)
Perfectly Auto Vectorized	66(43.71%)	82(54.31%)
Source Level Transformation	42(27.82%)	38(25.17%)
Using Intrinsics	16(10.6%)	7(4.64%)

Perfectly auto vectorized ≡ speedup > 1.15

Method	XLC	ICC	GCC
Auto Vectorization	1.66	1.84	1.58
Transformations	2.97	2.38	
Intrinsics	3.15	2.45	

Transformation Required	XLC	ICC	GCC
Total	58	45	70
if-conversion	10	0	14
Vectorization of `While` and `GOTO` loops	3	0	0
Algorithm substitution and Node Splitting	9	8	6
Statement Reordering	0	3	3
Data Copying	9	8	12
Wrap Around Variable Detection	5	2	5
Reduction	4	5	10
Rerolling	0	4	1
Symbolic Resolution	3	3	3
Loop Interchanging	3	2	5
Loop Distribution	1	1	2
Other	11	9	9

	XLC-ICC	ICC-XLC	ICC-GCC	GCC-ICC	XLC-GCC	GCC-XLC
Total	9	25	36	13	19	12
Statement Reordering	3	0	1	2	2	0
Loop Interchange	0	1	3	0	2	0
Rerolling	3	0	0	2	1	0
Reduction	1	2	6	1	4	0
if-conversion	0	9	14	0	9	3
Wrap Around Variable Detection	0	3	3	0	0	0
Data Copying	0	4	4	3	0	3
Other	2	6	5	5	1	6

- Overall, the compilers vectorize few cases of the loops in the applications in PACT and Media Bench II. Out of 33 loops
  - XLC vectorized 6 (18.18%),
  - ICC vectorized 10 (30.30%)
  - GCC vectorized 3 (10.0%).

Method	XLC	ICC	GCC
Auto Vectorization	1.154	1.279	1.232
Manual	2.101	2.743	2.692

Ongoing work

- Developing vector seeker to automate (partially) identification of compiler limitations.
- Another academic push on autovectorization ?

Autovectorization impossible ?

- It is not a matter of yes or no, but of degrees.
- ALL compilers autovectorize and much effort goes into that.

Why not ?


IBM
GARRY
KASPAROV
DEEP
BLUE


THINK
$300,000
$200,000
I'll take
Autovectorization for
$500, Alex

Conclusion

- Programming systems for parallelism still evolving. Much more to do.
- The problem has proved to be more complex than expected, but much progress has been made.
- Can only hope for a much better picture at the 50th LCPC