Programming in the Era of Parallelism

David Padua
University of Illinois at Urbana-Champaign
Outline of the talk

I. Introduction

II. Languages

III. Automatic program optimization
   • Compilers
   • Program synthesizers

IV. Conclusions
I. Introduction (1): The era of parallelism

• Their imminence announced so many times that it started to appear as if it was never going to happen.

• But it was well known that this was the future.

• This hope for the future and the importance of high-end machines led to extensive software activity from Illiac IV times to our days (with a bubble in the 1980s).
I. Introduction (2): Accomplishments

• Parallel algorithms.
• Widely used parallel programming notations
  – distributed memory (SPMD/MPI) and
  – shared memory (pthreads/OpenMP).
• Compiler and program synthesis algorithms
  – Automatically map computations and data onto parallel machines/devices.
  – Detection of parallelism.
• Education.
I. Introduction (3): Accomplishments

• Goal of architecture/software studies: to reduce the additional cost of parallelism.
  – Want efficiency/portable efficiency
The challenge of parallel programming

- Correctness.
  - Communication/synch errors → races and deadlocks.
- Portability – maintaining correctness and efficiency.
  - There will be a wider range of possibilities than in the sequential era. Heterogeneous machines.
- Scalability – Performance gains with each new generation
  - The free ride of faster clock rates is no more.
  - Lack of scalability more apparent than in the sequential era.
  - Scalability is the business model.
I. Introduction (4):
Present situation

• But much remains to be done and, most likely, widespread parallelism will give us performance at the expense of a dip in productivity.
I. Introduction (5):
The future

• Although advances not easy, we have now many ideas and significant experience.
• And … Industry interest → more resources to solve the problem.
• The extensive experience of massive deployment will also help.
• The situation is likely to improve rapidly. Exciting times ahead.
Outline of the talk

I. Introduction

II. Languages

III. Automatic program optimization
   • Compilers
   • Program synthesizers

IV. Conclusions
II. Languages (1):
OpenMP and MPI

- 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”.
- Alternatives have been designed. Next: an example of higher level language for numerical computing.
II. Languages (2): Hierarchically Tiled Arrays

- 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..

Joint work with IBM Research.
II. Languages (3): Hierarchically Tiled Arrays

2 X 2 tiles map to distinct modules of a cluster

4 X 4 tiles
Use to enhance locality on L1-cache

2 X 2 tiles map to registers
II. Languages (4): Accessing HTAs

tiles

$h\{1,1:2\}$

$h\{2,1\}$

hierarchical
II. Languages (5):
Tiled matrix multiplication

for I=1:q:n
    for J=1:q:n
        for K=1:q:n
            for i=I:I+q-1
                for j=J:J+q-1
                    for k=K:K+q-1
                        C(i,j)=C(i,j)+A(i,k)*B(k,j);
                    end
                end
            end
        end
    end
end

for i=1:m
    for j=1:m
        for k=1:m
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
II. Languages (6):
Higher level operations

- `repmat(h, [1, 3])`
- `circshift(h, [0, -1])`
- `transpose(h)`
II. Languages (7): Higher level operations

Operations implemented as messages if HTA is distributed
II. Languages (8):
User-defined operators

\[
\text{output} = \text{map}( \text{op}, \text{input} );
\]

<table>
<thead>
<tr>
<th>op</th>
<th>op</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>op</td>
<td>op</td>
</tr>
<tr>
<td>op</td>
<td>op</td>
<td>op</td>
</tr>
</tbody>
</table>

input

\[
\text{output} = \text{mapReduce}( \text{op}, \text{input} );
\]

output
**II. Languages (9): Cannon's parallel matrix multiplication**

<table>
<thead>
<tr>
<th></th>
<th>A₀₀</th>
<th>A₀₁</th>
<th>A₀₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁₀</td>
<td>A₁₁</td>
<td>A₁₂</td>
<td></td>
</tr>
<tr>
<td>A₂₀</td>
<td>A₂₁</td>
<td>A₂₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B₀₀</th>
<th>B₀₁</th>
<th>B₀₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁₀</td>
<td>B₁₁</td>
<td>B₁₂</td>
<td></td>
</tr>
<tr>
<td>B₂₀</td>
<td>B₂₁</td>
<td>B₂₂</td>
<td></td>
</tr>
</tbody>
</table>

**Initial skew**

**Shift-multiply-add**
II. Languages (10):
Cannon's parallel matrix Multiplication

```matlab
% Main loop
for i = 1:n
    c = c + a * b;
    a = circshift(a, [0, -1]);
    b = circshift(b, [-1, 0]);
end
```
II. Languages (11):
Summa matrix multiplication

function C = summa (A, B, C)
    for k=1:m
        T1 = repmat(A(:, k), 1, m);
        T2 = repmat(B{k, :}, m, 1);
        C = C + matmul(T1{:,:} ,T2 {:,:});
    end
II. Languages (12):
Advantages of tiles as first class objects

- Array/Tile notation produces code more readable than MPI. It significantly reduces number of lines of code.
II. Languages (13):
Advantages of tiles as a first class objects

Lines of Code. HTA vs. MPI

<table>
<thead>
<tr>
<th></th>
<th>HTA</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
II. Languages (14):
Advantages of tiles as first class objects

• More important advantage: Tiling is explicit. This simplifies/makes more effective automatic optimization.

```matlab
for i=1:m
    for j=1:m
        for k=1:m
            C{i,j}=C{i,j}+A{i,k}*B{k,j};
        end
    end
end
```

Size of tiles?
II. Languages (15):
Data parallelism beyond arrays:
Operations on aggregates

• Operations on aggregates do not have to be confined to arrays
• Other objects such as trees, graphs, and sets can and have been used in the past.
• A good example of the use of sets for programming is the language SETL.
II. Languages (16): Parallel search algorithm
II. Languages (17):
Parallel search algorithm

\[ W = \{ \text{root} \}; \]
\[ S = \text{solutions}(W) \]
\[ \text{while } S = \emptyset \]
\[ \text{ALL} = \text{expand}(W) \]
\[ W = \text{select}(\text{ALL}) \]
\[ S = \text{solutions}(W) \]
\[ \text{Tree} = \text{Tree} + \text{ALL} \quad \text{/* + is set union */} \]
II. Languages (18):
What problem domains and set operations

• We have studied several areas including
  – Search algorithms for discrete optimization
  – Datamining
  – Triangularization
• In all cases, it was possible to obtain a highly parallel version using set operations
II. Languages (18): Data parallel operators and parallel programming

- Parallel programs written based on operators resemble conventional, serial programs.
  - Parallelism is *encapsulated*.
  - Parallelism is *structured*.
- Parallelism could be *portable* across classes of machines.
  - Operations must be reimplemented for each new class of machine.
- Synthesis is possible
- Compiling facilitated by the higher level notation.
II. Languages (19):
Conclusions: What next?

• High-level notations/new languages should be studied. Much to be gained.
• Much potential in data parallel operations.
• But .. New languages by themselves will not go far enough in reducing costs of parallelization.
• Automatic optimization is needed.
• Parallel programming languages should be automatic optimization enablers.
  – Need language/compiler co-design.
Outline of the talk

I. Introduction

II. Languages

III. Automatic program optimization
   • Compilers
   • Program synthesizers

IV. Conclusions
III. Automatic Program Optimization (1)

• The objective of compilers from the outset.

“It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.”

John Backus
Fortran I, II and III
III. Automatic Program Optimization (2)

• Still far from solving the problem. CS problems seem much easier than they are.

• Two approaches:
  – Compilers
  – The emerging new area of program synthesis.
Outline of the talk

I. Introduction
II. Languages
III. Automatic program optimization
   • Compilers
   • Program synthesizers
IV. Conclusions
III.1 Compilers (1)
Purpose

• 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?
III.1 Compilers (2)
How well do they 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.
III.1 Compilers (3)
How well do they work?
Automatic detection of parallelism

III.1 Compilers (4)

How well do they work?

Vectorization

<table>
<thead>
<tr>
<th>Speedups</th>
<th>Manual Vectorization</th>
<th>ICC 8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculation of 4th LTP Filter</td>
<td>2.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Short Term Analysis Synthesis</td>
<td>3.0</td>
<td>1.5</td>
</tr>
<tr>
<td>gch_sieve2</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>synth.dat</td>
<td>2.0</td>
<td>1.25</td>
</tr>
<tr>
<td>jpeg.dct_slov</td>
<td>1.5</td>
<td>1.25</td>
</tr>
<tr>
<td>form_spectral prediction</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>idct</td>
<td>2.5</td>
<td>1.0</td>
</tr>
<tr>
<td>idct</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>jwPSmpout</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>jw accelerated triangle</td>
<td>2.0</td>
<td>1.25</td>
</tr>
<tr>
<td>gl_depth_test</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>mix_mystery signal</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

III. 1 Compilers (5)
How well do they work?
Locality enhancement

Matrix-matrix multiplication on Intel Xeon

60X

Intel MKL (hand-tuned assembly)

Is Search Really Necessary to Generate High-Performance BLAS?
Compiler vs. Manual Tuning
Matrix Matrix Multiplication

- Intel MKL
- icc -O3 -xT
- icc -O3

MFLOPS vs. Matrix Size

20x improvement
Compiler vs. Manual Tuning
Matrix Matrix Multiplication

loop 1
\[ c[i*N+j] += a[i*N+k]*b[k*N+j] \]

loop 2
\[ c[i][j] += a[i][k]*b[k][j] \]

loop 3
\[ C += a[i][k]*b[k][j] \]
III. 1 Compilers (6)
How well do they work?
Scalar optimizations

III. 1 Compilers (7)
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 (*and of course better languages*)
III.1 Compilers (8)  
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.
III.1 Compilers (9)
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.
III. 1 Compilers (10)

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”
Outline of the talk

I. Introduction
II. Languages
III. Automatic program optimization
   • Compilers
   • Program synthesizers
IV. Conclusions
III.2 Program Synthesizers (1)

- Emerging new field.
- Goal is to automatically generate highly efficient code for each target machine.
- Typically, a generator is executed to empirically search the space of possible algorithms/implementations.

- Examples:
  - In linear algebra: ATLAS, PhiPAC
  - In signal processing: FFTW, SPIRAL
III.2 Program Synthesizers (3)

• Automatic generation of libraries would
  – Reduce development cost
  – For a fixed cost, enable a wider range of implementations and thus make libraries more usable.

• Advantage over compilers: Can make use of semantics
  – More possibilities can be explored.

• Disadvantage over compilers: Domain specific.
III.2 Program Synthesizers (2)

Algorithm description

Generator / Search space explorer

High-level code

Source-to-source optimizer

Selected code

High-level code

Native compiler

Object code

Execution

Performance

Input data (training)
III.2 Program Synthesizers (4)

Three synthesis projects

1. **Spiral**
   
   Joint project with CMU and Drexel.
   

2. **Analytical models for ATLAS**
   
   Joint project with Cornell.
   

3. **Sorting and adaptation to the input**
   
   In all cases results are surprisingly good. Competitive or better than the best manual results.
III.2 Program Synthesizers (5)
Sorting routine synthesis

- During training several features are selected influenced by:
  - Architectural features
    - Different from platform to platform
  - Input characteristics
    - Only known at runtime
  - Features such as: Radix for sorting, how to sort small segments, when is a segment small.

X. Li, M. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. CGO2005
III.2 Program Synthesizers (6)
Sorting routine synthesis
Performance on Power4

![Graph showing performance (keys per cycle) vs. standard deviation for different libraries: C++ STL, XSort, IBM ESSL.](image)
III.2 Program Synthesizers (7)
Parallel sorting routine synthesis

Similar results were obtained for parallel sorting.

B. Garber. MS Thesis. UIUC. May 2006
III.2 Program Synthesizers (8)
Parallel sorting routine synthesis

Comparisons (Size: 100M keys)

- Cilksort
- Tuned Sort
- TBB QuickSort
- Typical Radix Sort

Time (seconds)

Standard Deviation (2^n)
III.2 Program Synthesizers (9)

Programming synthesizers

- Objective is to develop language extensions to implement parameterized programs.
- Values of the parameters are a function of the target machine and execution environment.
- Program synthesizers could be implemented using autotuning extensions.

III.2 Program Synthesizers (10)
Programming synthesizers
Example extensions.

#define search (1<=m<=10, a)
#define unroll m

for(i=1;i<n;i++) { ... }

if (a) then {algorithm 1}
else {algorithm 2}
III.2 Program Synthesizers (11)

Research issues

• Reduction of the search space with minimal impact on performance. Analytical models/avoiding search.

• Adaptation to the input data (not needed for dense linear algebra)

• More flexible synthesizers
  – algorithms
  – data structures
  – classes of target machines
Research issues

- Autotuning libraries.
  - Algorithms
  - Data parallel primitives
  - Empirically identified patterns or codelets

- Programming environments to facilitate development of synthesizers.
IV. Conclusions

• Advances in languages and automatic optimization will probably be slow. Difficult problem.
• Advent of parallelism → Decrease in productivity. Higher costs.
• But progress must and will be made.
• Automatic optimization (including parallelization) is a difficult problem. At the same time is a core of computer science:
  How much can we automate?
Acknowledgements

• I gratefully acknowledge support from DARPA ITO, DARPA HPCS program and NSF NGS program.