High level abstractions for irregular computations

David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
1. INTRODUCTION: THE PROBLEM
Programming for performance

• Is laborious
  – Particularly so when the target machine is parallel.
    • Must decide what to execute in parallel
    • In what order to traverse the data
    • How to partition the data ...  

• Today, tuning must be done manually.

• Maintenance is difficult
  – and the result is not portable.
Programming for performance

• There is still much room for progress in high performance programming methodology.
• Our goal is to facilitate (automate) tuning and enable machine independent programming.
2. IRREGULAR COMPUTATIONS
What are irregular computations?

• Irregular computations are difficult to define.

• One possible definition is:
  – Computations on “irregular” data structures.
  – Irregular data structures = not dense arrays.
    • Pingali et al. *The tao of parallelism in algorithms.* PLDI’11
  – And subscript expressions non-linear.
What are irregular computations?

• Examples include
  – Subscripted subscripts:
    • A[C[i]]
  – Linked lists traversal:
    • leftCell = leftCell->ColNext;
    • result+=leftCell->Value;
Programming irregular computations for performance

- Today, programming of irregular computations is typically at a low level of abstraction.

```fortran
  do diag=1,Nj, 2  ! two-way unroll amd Jam
    diaglen=min( ( jd_ptr(diag+1)-jd_ptr(diag) ) , \ 
                 ( jd_ptr(diag+2)-jd_ptr(diag+1) ) )
    offset1  = jd_ptr(diag) - 1
    offset2  = jd_ptr(diag+1) - 1
    do i=1, diagLen
      C(i) = C(i) + val(offset1+i)*B(col_idx(offset1+i))
      C(i) = C(i) + val(offset2+i)*B(col_idx(offset2+i))
    end do
    ! peeled off iterations
    offset1 = jd_ptr(diag)
    do i=(diagLen+1), (jd_ptr(diag+1)-jd_ptr(diag))
      C(i) = C(i)+val(offset1+i)*B(col_idx(offset1+i))
    end do
  end do
```

G. Hager and G. Wellein. Introduction to High Performance Computing for Scientists and Engineers. CRC Press
Programming irregular computations for performance

- Programming and optimizing irregular computations more challenging than for linear-subscripts computations.

- Want to have compiler support for programming at a low level of abstraction to enable
  - Automatically mapping computation onto diverse machine classes
  - Locality enhancement
  - Other compiler optimizations (e.g. common subexpression eliminations)

- In addition, often using higher levels of abstractions and applying optimizations at that level is a better solution.

- Both approaches will be discussed in this presentation
When computations are not irregular

• Compiler can do a good job at transforming programs:
  – Loop interchange
  – Loop distribution
  – Tiling
  – ...

• And we have the technology to generate efficient code for a variety of platforms even when the initial code is sequential
  – Vector extensions
  – Multicores
  – Distributed memory machines
  – GPUs
Example
Automatic transformation of a regular computation

• Consider the loop

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j]=a[i][j-1]+a[i-1][j];
    }
}
```
Example

Automatic transformation of a regular computation

- Compute dependences (part 1)

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

Values:
- \( a[1][1] = a[1][0] + a[0][1] \)
- \( a[1][2] = a[1][1] + a[0][2] \)
- \( a[1][3] = a[1][2] + a[0][3] \)
- \( a[1][4] = a[1][3] + a[0][4] \)
- \( a[2][1] = a[2][0] + a[1][1] \)
- \( a[2][2] = a[2][1] + a[1][2] \)
- \( a[2][3] = a[2][2] + a[1][3] \)
Example
Automatic transformation of a regular computation

• Compute dependences (part 2)

```java
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

<table>
<thead>
<tr>
<th>i=1</th>
<th>i=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=1</td>
<td>a[1][1] = a[1][0] + a[0][1] a[2][1] = a[2][0] + a[1][1]</td>
</tr>
<tr>
<td>j=2</td>
<td>a[1][2] = a[1][1] + a[0][2] a[2][2] = a[2][1] + a[1][2]</td>
</tr>
<tr>
<td>j=3</td>
<td>a[1][3] = a[1][2] + a[0][3] a[2][3] = a[2][2] + a[1][3]</td>
</tr>
</tbody>
</table>

December 4, 2017
Example
Automatic transformation of a regular computation

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j]=a[i][j-1]+a[i-1][j];
    }
}
```

December 4, 2017
Example
Automatic transformation of a regular computation

- Find parallelism

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
Example
Automatic transformation of a regular computation

• Transform the code

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j]=a[i][j-1]+a[i-1][j];
    }
}
```

```c
for (k=4; k<2*n; k++)
    forall (i=max(2,k-n):min(n,k-2))
    a[i][k-i]=...;
```
Analyzing irregular computations

• Analysis of the following code is difficult and sometimes impossible

\[
\text{for } (i=1; i<n; i++) \{ \\
\quad a[c[i]] = a[d[i]] + 1;
\}
\]

• Dependences are a function of the values of \(c\) and \(d\)
• In the absence of additional information, the compiler must assume the worst.

• And this precludes automatic transformations (parallelization, vectorization, data distribution, ...)

December 4, 2017
Irregularity is a beast programming language researchers have been fighting for a long time
It has defeated us sometimes

Ignoring the importance of irregular computations was one of the Reasons for the failure of High Performance Fortran
Taming the irregular beast

- Compilers and Runtime (Section 3)
- Better notations (Section 4)
3. COMPILER AND RUNTIME SOLUTIONS FOR LOW LEVEL IRREGULAR PROGRAMMING
Two approaches to dealing with irregular computations

• Analyzing irregular codes directly
  – Statically
  – Dynamically

• Converting into higher level of abstraction (dense array computations)
Static analysis

• In some cases, it is possible to analyze irregular algorithms written in conventional notation.

• More effort than for regular/linear subscript computations
Static analysis

\[
\begin{align*}
\text{do } k &= 1, \ n \\
q &= 0 \\
\text{do } i &= 1, \ p \\
\quad \text{if } ( x(i) > 0 ) \ \text{then} \\
\quad q &= q + 1 \\
\quad \text{ind}(q) &= i \\
\quad \text{end if} \\
\text{end do} \\
\text{do } j &= 1, \ q \\
\quad x(\text{ind}(j)) &= x(\text{ind}(j))* y(\text{ind}(j)) \\
\text{end do} \\
\text{end do}
\end{align*}
\]
do k = 1, n
    q = 0
    do i = 1, p
        if ( x(i) > 0 ) then
            q = q + 1
            ind(q) = i
        end if
    end do
end do

do j = 1, q
    x(ind(j)) = x(ind(j))* y(ind(j))
end do
end do

Need information about ind(j)

Dependence analysis
do k = 1, n
  q = 0
  do i = 1, p
    if ( x(i) > 0 ) then
      q = q + 1
      ind(q) = i
    end if
  end do
end do

do j = 1, q
  x(ind(j)) = x(ind(j))* y(ind(j))
end do

Need information about \texttt{ind(j)}
Can be found by analyzing the program
do i=1, n
  do j=2, iblen(i)
    do k=1, j-1
      x(pptr(i)+k-1) = ...
    end do
  end do
end do

do j=1, iblen(i)-1
  do k=1, j
    ... = x(iblen(i)+pptr(i)+k-j-1)
  end do
end do
end do

Array privatization
Array privatization

To decide if $x$ can be privatized

\begin{verbatim}
\begin{verbatim}
do i=1, n 
  do j=2, iblen(i) 
    do k=1, j-1 
      x(pptr(i)+k-1) = ... 
    end do 
  end do 
  do j=1, iblen(i)-1 
    do k=1, j 
      ... = x(iblen(i)+pptr(i)+k-j-1) 
    end do 
  end do 
end do
\end{verbatim}
\end{verbatim}
Static analysis

```plaintext
do i=1, n
  do j=2, iblen(i)
    do k=1, j-1
      x(pptr(i)+k-1) = ...
    end do
  end do
end do

do j=1, iblen(i)-1
  do k=1, j
    ... = x(iblen(i)+pptr(i)+k-j-1)
  end do
end do
```

To decide if $x$ can be privatized is sufficient to show that a write precedes each read.

Array privatization
Static analysis

• A possible approach is to query the control flow graph, bounding the search.

Yuan Lin and David Padua. Compiler analysis of irregular memory accesses. (PLDI '00).
Dynamic analysis

- Embedded dependence analysis
- Inspector executor
- Speculation
Embedded dependence analysis

• This is a clever mechanism due to Zhu and Yew.

```
for i=1; i<n; i++
a(k(i)) = c(i) + 1
```

```
parallel for i=1;i<n;i++
critical a(k(i))
if a_tag(k(i)) < i
  a(k(i)) = c(i) + 1
  a_tag(k(i)) = i
```

Inspector executor

- Parallel schedule and communication aggregation computed at execution time
  - Analyzing the memory access pattern of a code segment (typically a loop)
- Overhead reduced when the code segment (loop) is executed multiple times with the same access pattern

See Encyclopedia of Parallel Computing
  Run Time Parallelization (Saltz and Das)
Speculation

• There is an extensive body of literature on speculation.
• A code segment is executed in parallel in the hope that there is no problem.
• If there is, execution backtracks and the code segment is re-executed serially


Encyclopedia of Parallel Computing
  Thread level speculation (Torrellas)
  Transactional Memories (Herlihy)
Converting into dense array computations

• The objective is to convert sparse computations into equivalent dense form.
• This increases the number of operations involving zeroes (or identity)
  – Sometimes making the computation impossible
• But it allows the application of compile-time transformations.
• Sparsity is recovered by applying a final transformation (see below)
Converting into dense array computations

```c
for(col=0; col<cols; col++) {
    for(row=0; row<dimensions; row++) {
        leftCell = left.Rows[row];
        while( leftCell != NULL ) {
            result[row][col] += leftCell->Value
                                * right[leftCell->ColIndex][col];
            leftCell = leftCell->ColNext;
        }
    }
}
```

```c
for( col = 0; col < cols; col++ )
    for( row = 0; row < dimensions; row++ )
        result[row][col] += A_Valuep[row][:] * right[ :][col];
```
Converting into dense array computations

Harmen L. A. van der Spek, Harry A. G. Wijshoff: Sublimation: Expanding Data Structures to Enable Data Instance Specific Optimizations. LCPC 2010: 106-120

Where do we stand?

• There is an extensive body of literature on compilation and runtime to facilitate the parallel programming of irregular computations.

• Although more ideas are important, what we need most urgently is an evaluation of their effectiveness.
  – A means for refinement and progress.

• All compiler techniques suffer from lack of understanding of their effectiveness.
Need tools to evaluate progress

• Measuring advances in compiler technology is crucial for progress.

• An idea: a publicly available repository
  – Containing extensive and representative code collections.
  – Keep track of results for compilers/techniques along time.

• Ongoing work on one such repository:
4. NOTATIONS
4. NOTATIONS

4.1 ABSTRACTION
What are Programming abstractions?

- An abstraction is formed by reducing information.
- In everyday life, it forms the world of ideas where we can reason. No (natural language) statement can be made without abstractions.
What are Programming abstractions?

- Abstract *machine language/lower level* implementation
  - Scalar computations:
    - \[ a = b \times c + d^3.4 \]
    - No loads, stores, register allocation.
  - Scalar loops:
    - `for (i = ....) {...}
    - No if statements, branch/goto instructions.
    - Gives structure to the program.

- Abstract algorithm implementation
  - `min(A(1:n))`
  - `it = find (myvec.begin(), myvec.end(), 30);`
  - Hide algorithm, data representation
What are the benefit of using abstractions?

- **Programmer productivity (expressiveness)**
  - Codes are easier to write
  - To understand, maintain

- **Portability**

- **Optimization (manual and automatic)**
  - Programs are easier to analyze
  - Easier to transform
  - Deeper transformations are enabled
Abstractions and irregular computations

• What abstractions should we use for programming irregular computations at a higher level?

• Best answer seems to be to represent irregular algorithms in terms of:

  Dense array operations
4. NOTATIONS

4.2. ARRAY NOTATION AND IRREGULAR COMPUTATIONS
Why array notation

• Could access arrays in scalar mode
• But operations on aggregates are (often) easier to read.

\[
\begin{align*}
\text{for } (i=0; i<n; i++) & \text{ for } (j=0; j<n; j++) \\
a[i][j] &= b[i][j] + c[i][j]
\end{align*}
\]

\[
a = b + c
\]

• They are well understood abstractions.
• Powerful.
Array operations and performance

• But usefulness of array notation go beyond expressiveness.
• There are at least three reasons to use array abstractions for performance
  – To represent parallelism
  – To reduce the overhead of interpretation.
  – To enable automatic optimizations.
Representation of parallelism

• Popular in the early days:

```fortran
do 10 i = 1, 100, 2
   M (i) = .true.
   M (i+1) = .false.
10 continue
A(*) = B(*) + A(*)
C(M(*)) = B(M(*)) + A(M(*))
```

Illiac IV Fortran

• Used today but not widely

```c
C[i][ j] = __sec_reduce_add( a[ i][:] * b[:][j]);
```

Cilk plus

• Tensor flow
• Futhark
  — T. Henriksen, N. Serup, M. Elsman, F. Henglein, and C. Oancea. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates PLDI 2017
Array operations and optimizations
An example

• Applying arithmetic rules such as associativity of matrix-matrix multiplication to reduce the number of operations.
  – Which one is better?
    • (((A*B)*C)*D)*E,
    • (A*((B*C)*D))*E,
    • (A*B)*((C*D)*E),
    • ...
  – Find best using dynamic programming
Reduce overhead of interpretation

• See Haichuan Wang, David A. Padua, Peng Wu: Vectorization of apply to reduce interpretation overhead of R. OOPSLA 2015: 400-415
Array operations and irregular computations

• Array operations are an ideal notation to manipulate sparse arrays.
Sparse arrays in MATLAB

```matlab
load west0479.mat
A = west0479;
S = A * A' + speye(size(A));
pct = 100 / numel(A);
figure
spy(S)
title('A Sparse Symmetric Matrix')
nz = nnz(S);
xlabel(sprintf('nonzeros = %d (%.3f%%)',nz,nz*pct));
```

- Arrays appear as dense, but represented internally as sparse.
- The interpreter and libraries handle the sparseness
Sparse arrays can be abstracted as dense arrays in compiled languages

- Bik and Wijshoff* developed a strategy to translate dense array programs onto sparse equivalents.

\[
\begin{align*}
\text{do } i=1,n \\
\quad \text{acc}=\text{acc}+a(i,:)*<1:n> \\
\end{align*}
\]

\[
\begin{align*}
\text{do } i=1,n \\
\quad \text{do } j=1,n \\
\quad \quad \text{if } (i,j) \in E_a \text{ then} \\
\quad \quad \quad \text{acc}=\text{acc}+a'(\sigma_a(i,j))*j \\
\end{align*}
\]

\[
\begin{align*}
\text{do } i=1,n \\
\quad \text{do } \text{ad} \in \text{PAD}_a \\
\quad \quad j=\pi_2 \sigma_a^{-1}(\text{ad}) \\
\quad \quad \text{acc}=\text{acc}+a'(\text{ad})*j \\
\end{align*}
\]

\[
\begin{align*}
\text{do } i=1,n \\
\quad \text{do } \text{ad} = \text{allow}(i), \text{ahigh}(j) \\
\quad \quad j=\text{aind}(\text{ad}) \\
\quad \quad \text{acc}=\text{acc}+\text{aval}(\text{ad})*j \\
\end{align*}
\]


December 4, 2017
Array operations and irregular computations

• Not only can array notation be used to represent dense computations, it can be used for a number of other domains.

• A wide range of domains
At the time (ca. 1959), Illinois, with the first of its ‘ILLIAC’ supercomputers, was a great centre of computing, and Golub showed his affection for his Alma Mater by endowing a chair there 50 years later. Rumour has it that the funds for the gift came from Google stock acquired in exchange for some advice on linear algebra. Google’s PageRank search technology starts from a matrix computation — an eigenvalue problem with dimensions in the billions. Hardly surprising, Golub would have said: everything is linear algebra. Nature 2007
New operators
Arrays for graphs algorithms

• A simple algorithm for SSSP (Bellman-Ford) can be represented as follows:

\[
\text{for}(i=1; \ i<=n; \ i++) \\
\quad d = d \oplus A \otimes d
\]


Standards for Graph Algorithm Primitives
4. NOTATIONS

4.3. EXTENSIONS TO ARRAY NOTATION FOR PARALLELISM
Communication as array operations

- \texttt{repmat(h, [1, 3])}
- \texttt{circshift(h, [0, -1])}
- \texttt{transpose(h)}
Cannon’s algorithm

initial skew

shift-multiply-add

December 4, 2017
Cannon’s Algorithm

\[
\text{for } k=1:n \\
\quad c = c + a \times b; \\
\quad a = \text{circshift}(a,[0 -1]); \\
\quad b = \text{circshift}(b,[-1 0]); \\
\text{end}
\]

Barrier elimination

\[ X(1:4) = Y(1:4) \times 2; \]
\[ Z(1:2) = X([1, 3]); \]
\[ Z(3:4) = X([2, 4]); \]
Barrier elimination
Asynchronous semantics

• In some cases, it could be desirable to delay updates across the whole machine. At some point, a global update is made.

• This is a particular example of communication in asynchronous algorithms where data assignment suffer delays.
// the distance vector
d = inf; d(1) = 0;
// the mask vector
L = 0; L(1) = 1;
while (L.nnz())
    for i = 1:LOC_ITER
        r <- d + (trans(A)*(d*.L);
        L <- (r != d);
        d <- r;
    r += d;
    L <- (r != d);
    d <- r;

Saeed Maleki, G. Carl Evans, David A. Padua:
Tiled Linear Algebra a System for Parallel Graph Algorithms. LCPC 2014: 116-130
4. NOTATIONS

4.3. WHERE DO WE STAND?
Array notation

• Array notation has a long history
• Is a powerful notation that can be used for a wide range of applications.
• For parallelism, it is an unfulfilled promise
• The highly mathematical notation is a challenge
• We see increase use of the notation and expect it to become popular also for irregular algorithms.
• Challenges:
  – develop the notation
  – Design compiler optimizations
5. EPILOG
• In 1976 not much was known about parallel programming of irregular computations.
• 40 years later if is impressive how much have been learned on compilation and notations.
• But the development of widely used tools based on these ideas remain a challenge.