Automatic Programming

Robert van Engelen
Programming

• Is fun
• Is creative
• Is satisfying to solve real-world problems
Programming

• Is fun
• Is creative
• Is satisfying to solve real-world problems

• The old saying goes that programming is 80% transpiration and 20% inspiration
• “Mythical man month” suggests that effort does not scale with project size
• Parallel programming methodologies still in infancy stage
Lots of Tools for Code Checking

To name just a few:

**Static code analysis:**
- Lint and splint: GNU tool for bug-sniffing C
- PC-lint: bug-sniffing C++
- clang: C/C++ compiler and static checker

**Model checking** steps through every possible execution path:
- Klocwork: C/C++, C#, and Java analysis
- Coverity: C/C+ analysis

**Dynamic analysis**
- Valgrind: runtime memory, threading, and code analysis
- Dmalloc: malloc debugger
- Insure++: runtime memory error detection
- TotalView: profiling and debugging OpenMP/MPI apps
- Alinea DDT: profiling and debugging OpenMP/MPI/Cuda apps
Static Analysis vs. Model Checking vs. Formal Verification

Program Analysis
- Static Semantic Analysis
- Data Flow Analysis
- Abstract Interpretation
- Model Checking
- Axiomatic Semantics
- Denotational Semantics
- Operational Semantics
- Logical Inference
- Theorem Proving

Research Seminar 2011
Accurate and Fast Static Analysis is Essential for Compiler Optimizations

• Given a program $P$, produce a program $P'$ that produces the same output values as $P$ for a given input, but has a lower cost [Aho88]

• Optimization is an economic activity [Novak05]
  – Cost: a larger and sometimes slower compiler
  – Benefit: amount saved by the code improvement

• It is not possible to optimize everything
  – NP completeness and undecidability of many program analysis and optimization problems

• The goal is to find leverage: cases where there is a large expected payoff for a small cost
Compilers are Powerful Optimizers, but they’re Brain-Dead

Many inefficiencies can be optimized away by compilers if the static analysis and data flow analysis are sufficiently powerful.

Warning: compilers optimize code without regard to parallel execution!

```c
x = 1;  // compiler removes this as dead code
...
// because there is no use of x here
x = 0;
```
Compilers are Powerful Optimizers, but they’re Brain-Dead

Many inefficiencies can be optimized away by compilers if the static analysis and data flow analysis are sufficiently powerful.

Warning: compilers optimize code without regard to parallel execution!

Process 0

\[ x = 1; \] // removed
\[ ... \] // no use of x here
\[ x = 0; \]

Process 1

if \( (x == 1) \)

exit(0);

Note: use volatile in C
A Static Checker is your Friend!

What is the problem with this example that a compiler can warn about?

```plaintext
1  if( x != 0 )
2    if( p ) *p = *p/x;
3    else
4    if( p ) *p = 0;
```
A Static Checker is your Friend!

What is the problem that a compiler will spot here?

```c
3  void out( int n )
4  {
5      cout << n << "\n";
6  }
7
8  void show( int a, int b, int c )
9  {
10     out( a ); out( b ); out( c );
11  }
12
13  int main()
14  {
15      int i = 1;
16      show( i++, i++, i++ );
17      return 0;
18  }
```
A Static Checker is your Friend!

What is the problem that a compiler will spot here?

```c
unsigned a[100] = {0};

int main()
{
    char buf[200];
    unsigned n = 0;

    while( fgets( buf, 200, stdin ) )
    {
        if( n < 100 ) a[n++] = strlen(buf);

        while( --n >= 0 )
            printf( "%d\n", a[n] );
    }
    return 0;
}
```
A Static Checker is your Friend!

What is the problem that a compiler will spot here?

```c
3    void print_mod( int i, int n )
4    {
5        if( n == 0 && i == 0 ) return;
6        printf( "%d mod %d == %d\n", i, n, i % n );
7    }
8
9    int main()
10    {
11        for( int i = 0; i < 10; i++ )
12            for( int j = 0; j < 10; j++ )
13                print_mod( i, j );
14        return 0;
15    }
```
A Static Checker is your Friend!

What is the problem that a compiler will spot here?

```c
int shamrock_count( int leaves, double leavesPerShamrock ) {
    double shamrocks = leaves;
    shamrocks /= leavesPerShamrock;
    return leaves;
}

int main() {
    printf( "%d
", shamrock_count( 314159, 3.14159 ) );
    return 0;
}
```
A Static Checker is your Friend!
What is the problem that a compiler can spot here?

```ruby
1 class BankAccount
2 
3   def accountName
4     @accountName = "John Smith"
5   end
6 
7   def deposit
8     @deposit
9   end
10 
11   def deposit=(dollars)
12     @deposit = dollars
13   end
14 
15   def initialize ()
16     @deposet = 100.00
17   end
18 
19   def test_method
20     puts "The class is working"
21     puts accountName
22   end
23 end
```
Let’s Take a Look at Static Analysis

A novice programmer once wrote (true story):

```c
for( int i = 0; i < 5; i++ )
    p++;
```

He probably meant to increase the pointer `p` by 5
Let’s Take a Look at Static Analysis

```c
for( int i = 0; i < 5; i++ )
    p++;
```

Fortunately, a compiler with an excellent static analyzer will optimize this loop to `p += 5`

**Question:** can we do the same for the following loop with `n`?

```c
for( int i = 0; i < n; i++ )
    p++;
```
Let’s Take a Look at Static Analysis

```c
for( int i = 0; i < 5; i++ )
p++;
```

Fortunately, a compiler with an excellent static analyzer will optimize this loop to `p += 5`

**Question**: can we do the same for the following loop with `n`?

```c
for( int i = 0; i < n; i++ )
p++;
```

**Answer**: 
```c
if(n > 0)
p += n;
```
Formal Verification

How do we know **for sure** that the transformed code

```c
if(n > 0)
    p += n;
```

is correct?

We use **formal verification**: first we rewrite

```c
for( int i = 0; i < n; i++ )
    p++;
```

into the de-sugared equivalent form:

```c
int i = 0;
while( i < n ) {
    p = p + 1;
    i = i + 1;
}
```
Formal Verification with Axiomatic Semantics

\[
\{ 0 \leq n \land p = q \} \\
\{ 0 \leq n \land p - q = 0 \}
\]

\( i = 0; \)

\( \{ i \leq n \land p - q = i \} \)

while( \( i < n \) ) {

\( \{ i < n \land p - q = i \} \)

\( \{ i+1 \leq n \land p+1 - q = i+1 \} \)

\( p = p + 1; \)

\( \{ i+1 \leq n \land p - q = i+1 \} \)

\( i = i + 1; \)

\( \{ i \leq n \land p - q = i \} \)

}\)

\( \{ i \geq n \land i \leq n \land p - q = i \} \)

\( \{ p = q + n \} \)

weakest precondition

apply assignment rule

loop invariant

apply assignment rule

loop invariant

apply assignment rule

loop invariant

i \geq n \land loop invariant

postcondition
Formal Verification

• Axiomatic semantics assumes
  – a relatively simple machine model (state)
  – a relatively simple data type system (no pointers)
  – requires the user to determine invariants

• Not suitable for large projects
A Static Checker is your Friend!

What is the problem that the compiler can spot here?

```c
int a[10000];

void init()
{
    int i;
    for( i = 0; i <= 10000; i++ );
    a[i] = i;
}
```
How to Analyze: Abstract Interpretation

Define a lattice:
\[ [a,b] \sqcup [a',b'] = [\min(a,a'),\max(b,b')] \]
\[ [a,b] \sqcap [a',b'] = [\max(a,a'),\min(b,b')] \]

```c
int a[10000];
i = 0;
while (i <= 10000) {
    a[i] = i;
i = i + 1;
}
```

\[ p_0: \quad i = [0,0] \]
\[ p_1: \quad i = [0,0] \sqcap [-\infty,10000] = [0,0] \]
\[ p_2: \quad i = [0,0] \cup [1,1] \sqcap [-\infty,10000] = [0,1] \]
\[ \ldots \text{use } acceleration \text{ to determine finite convergence} \]
\[ i = [0,10000] \]

\[ p_1: \quad i = [1,1] \sqcap [-\infty,10001] = [1,1] \]
\[ p_2: \quad i = [1,1] \cup [2,2] \sqcap [-\infty,10001] = [1,2] \]
\[ \ldots \text{use } acceleration \text{ to determine finite convergence} \]
\[ i = [1,10001] \]

\[ p_3: \quad i = [1,10001] \sqcap [10001,\infty] = [10001,10001] \]
Limitations of Static Analysis

What about this example?

```
void init(int *a)
{
    int i;

    for( i = 0; i <= 10000; i++ )
    {
        a[i] = i;
    }
}
```
Model Checking

Verifies every execution path, along updating the program state

```
1 int n = 9999;
2 int *b = malloc(sizeof(int) * n);
   init(b+1);
3 void init(int *a)
4 {
5     int i;
6
7     for( i = 0; i <= 10000; i++ );
8     a[i] = i;
9 }
```
Model Checking

Still limited when program depends on runtime data

```cpp
int n; n << std::in;
int *b = new int[n];
init(b);

void init(int *a)
{
    int i;
    for (i = 0; i <= 10000; i++)
    a[i] = i;
}
```
Model Checking of Parallel Code

**Process 1**

\[
\begin{align*}
    p_0 & : \texttt{while} (x > 0) \{ \\
    p_1 & : \texttt{use resource} \\
    x & = x + 1; \\
    \}
    \\
    p_2 & : \texttt{sleep};
\end{align*}
\]

**Process 2**

\[
\begin{align*}
    q_0 & : x = 0; \\
    q_1 & : \texttt{use resource forever}
\end{align*}
\]

**Model**

(using abstract traces where \(x\) is positive or 0)

Q: starting with \(x > 1\), will \(p_1\) ever concurrently execute with \(q_1\)?

Q: will execution reach a state where \(x\) stays 0?
What did we Learn so far?

• Compilers use static analysis
  – for code optimization
  – for compiler warnings

• Static analysis has limitations
  – complexity of the code structure
  – the absence of parallel threads that change shared state
  – needs “knowns” (constants)

• Model checking is more comprehensive, applies to parallel code

• Formal verification is not automatic and costly
Can we do Better?

• Programming hasn’t changed since the 80s
• Cost of developing and maintaining code is high
• An industrial revolution is needed to scale it up
• Source code generation by automatic programming:

  the generation of source code by computer, usually based on specifications that are higher-level and easier for humans to specify than ordinary programming languages
Automatic Programming by Code Generation

• Specialization, aka lowering of code
  – From high-level abstract specifications
  – ... to intermediate high-level coding forms
  – ... to low-level Fortran, C/C++, ... whatever

• How?
  – By pattern matching and substitution, or some form of computer algebra
  – By type systems (a form of theorem proving) to propagate information and intelligently convert code
  – By high-level compiler static analysis and optimization
Automatic Programming by Specialization and Optimization

• Specialization by pattern matching and code rewriting lowers the abstract code to more concrete forms

\[ x[j] = \text{sum}(b \cdot a[i,j] + c, \ i=1..n) \quad \text{forall} \ j=1..m \]

doall(j, 1, m, x[j] = b \cdot \text{reduce}(a[j], 1, n)+c)

\$OMP \ DO \ PRIVATE(i,t) \ DO \ j=1,m \ t = c \ DO \ i=1,n \ t = t+a[i,j] \ ENDDO \ x[j] = b \cdot t \ ENDDO \ SOMP \ END \ DO

• Each lowering phase requires an optimization phase applicable to that code level
Publications Related to Automatic Programming

• Code generators and numerical applications


Publications Related to Automatic Programming

• Code generators and numerical applications (cont’d)


Publications Related to Automatic Programming

- Code generators for XML Web services protocol implementations, such as SOAP, WSDL, WS-*
Publications Related to Static Analysis

- Array dependence testing, induction variables and loop scalar evolutions (used in GCC 4.x !)
  
  
  
  
  
  
  
  - Robert A. van Engelen, Efficient Symbolic Analysis for Optimizing Compilers, in the proceedings of the International Conference on Compiler Construction, ETAPS 2001, LNCS 2027, pages 118-132
Publications Related to Formal Verification

- Automatic verification of code-improving transformations


Example: Automatic Programming for Atmospheric Models

Compute for each grid point in 3D space:

- Pressure \( p \)
- Temperature \( T \)
- Humidity \( q \)
- Wind velocity vector \( (u,v) \)
Numerical Code Generation

High-level equation

\[ p = \int_{0}^{1} \int_{y}^{1} \frac{\partial u}{\partial x} \, dy \, dz \quad \forall (x, y) \in \Omega_{x,y} \quad (1) \]

Transform, optimize

Parallel code

```fortran
PROGRAM P3
REAL u(0:n+1,m,1),q(0:n+1),h,p(0:n+1,m),q(0:n+1)
REAL s(0:n+1),t(0:n+1,m)
...
CMIC$ PARALLEL ...
CMIC$ CASE
DO 2300 j = 1,m
    FORALL(i=0:n+1) s(i)=s(i)+u(i,j,1)
2300 CONTINUE
CMIC$ CASE
    DO 2310 j = m,2,-1
        FORALL(i=1:n+1) t(i,j-1)=t(i,j)
    2310 CONTINUE
CMIC$ PARALLEL ...
CMIC$ CASE
    FORALL(i=1:n) q(i)=g(i)*(s(i)-s(i-1))/h
CMIC$ CASE
    FORALL(i=1:n+1,j=1:m) t(i,j)=t(i,j)/h
CMIC$ END CASE
CMIC$ END PARALLEL
FORALL(i=1:n,j=1:m) p(i,j)=t(i+1,j)-t(i,j)
```
Step 1 (Temperature $T$)

\[ T_i := \left\{ \begin{array}{l}
\left( \begin{array}{c}
T \\
\frac{1}{h_x h_y} \left( g \left( \frac{1}{c} - 1 \right) + 1 \right)
\end{array} \right) \\
\left( \begin{array}{c}
\frac{\partial (h_y v_{aux})}{\partial p} + \frac{\partial (h_x v_{aux})}{\partial y} \\
\frac{\partial \log(p)}{\partial z} [1] \\
-h_y v_{aux} T \left( q \left( \frac{1}{c} - 1 \right) + 1 \right)
\end{array} \right)
\end{array} \right\} \\
\left( \begin{array}{c}
0 \\
1 - \left( \frac{\partial \log(p)}{\partial z} [1] \right) \\
-0.693147180559945
\end{array} \right) \\
\left( \begin{array}{c}
\frac{\partial \log(p)}{\partial z} [1] \\
0.693147180559945 \\
\int_{k+1}^{nlev+1} \left( \frac{1}{h_x h_y} \left( \frac{\partial (h_y v_{aux})}{\partial x} + \frac{\partial (h_x v_{aux})}{\partial y} \right) \right) dz [1] \\
\left( \frac{\partial (g \left( \frac{1}{c} - 1 \right) + 1)}{\partial \eta} \right) \left( \frac{\partial p}{\partial \eta} \right) \left( q \left( \frac{1}{c} - 1 \right) + 1 \right)
\end{array} \right) \\
- \left( \begin{array}{c}
\frac{\partial \eta}{\partial \eta} \left( q \left( \frac{1}{c} - 1 \right) + 1 \right)
\end{array} \right)
\right\} \\
\left( \begin{array}{c}
\frac{\partial p}{\partial \eta} \\
- \left( h_y v_{aux} \frac{\partial T}{\partial x} + h_x v_{aux} \frac{\partial T}{\partial y} \right)
\end{array} \right)
\]
Step 2 (Temperature $T$)

$$T_x := \frac{1}{h_x h_y} \left[ \begin{array}{c}
\left( q \left( \frac{1}{z} - 1 \right) + 1 \right) \\
\left( \frac{\partial T}{\partial x} \right) \\
\frac{\partial (h_x \cdot u_{ax})}{\partial x} \\
\frac{\partial (h_y \cdot u_{ay})}{\partial y}
\end{array} \right] + \left[ \begin{array}{c}
\left( \frac{\partial (h_x \cdot u_{ax})}{\partial x} \right) \\
\left( \frac{\partial (h_y \cdot u_{ay})}{\partial y} \right) \\
\frac{\partial (h_x \cdot u_{ax})}{\partial x} + \frac{\partial (h_y \cdot u_{ay})}{\partial y}
\end{array} \right] \left( \begin{array}{c}
p_x T \\
\frac{\partial (h_x \cdot u_{ax})}{\partial x} \\
\frac{\partial (h_y \cdot u_{ay})}{\partial y}
\end{array} \right) \left( \int_{z_{x+1}}^{z_{x+2}} \left( \frac{\partial (h_x \cdot u_{ax})}{\partial x} + \frac{\partial (h_y \cdot u_{ay})}{\partial y} \right) dx \right)$$

$$\frac{\partial T}{\partial z} \left\{ \begin{array}{l}
\frac{\partial T}{\partial z} \\
\frac{\partial (h_x \cdot u_{ax})}{\partial x} + \frac{\partial (h_y \cdot u_{ay})}{\partial y}
\end{array} \right\} \left( \begin{array}{c}
\left( \frac{1}{z} - 1 \right) + 1 \\
\left( \frac{\partial T}{\partial x} \right) \\
\frac{\partial (h_x \cdot u_{ax})}{\partial x} \\
\frac{\partial (h_y \cdot u_{ay})}{\partial y}
\end{array} \right)$$

Research Seminar 2011
Step 3 (Temperature $T$)

\[
T_t := \begin{pmatrix}
\frac{1}{\Delta_z T} \left\{ \eta \frac{\partial p}{\partial v} \right\}^{\frac{1}{2}} \\
\Delta_z T
\end{pmatrix}
\]

Research Seminar 2011

40
Step 4

CMIC$ DO ALL
DO 1120 k = 1,nlev
DO 1130 j = 1,nlat
DO 1140 i = 1,nlon
IF (k.LE.1) THEN
  PDTDT(i,j,k) = (0.5*t144(i,j,2)+(1.55424620615724E-8*
    PTZ(i,j,k)*(1/(0.860825768434464*PQZ(i,j,k)+1)+0.60782469342251*
    rdth*(hxv(i,j)*t49(i,j,k)*PTZ(i,j,k)-hxv(i,j-1)*t49(i,j,k-1)*PVZ(i,j,k-1)))+
    9*(PQZ(i,j,1)/((0.860825768434464*PQZ(i,j,1)+1)*rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdlam*(hyu(i,j)*t47(i,j,1)-hyu(i-1,j)*t47(i-1,j,1)*PUZ(i-1,j,1)+
    rdth*(hxv(i,j)*t49(i,j,k-1)*PVZ(i,j,k-1)-hxv(i,j-1)*t49(i,j,k-2)*PVZ(i,j,k-2)))+
    Z(i,j,1)))+7.84806152880239E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    1,1)+rdth*((-t120(i,j,k)-t120(i,j,k+1)))+0.285714285714286*(PTZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+0.607824693422519*(PQZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+6.30915356075388E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    (0.860825768434464*PQZ(i,j,1)+1))/(hxt(i,j)*hyt(i,j))/(p(i,j,k+1)-p(i,j,k))
ELSE IF (nlev.LE.k) THEN
  PDTDT(i,j,k) = (0.285714285714286*PTZ(i,j,k)*(7.84806152880239E-8*
    Z(i,j,1)+t18(i,j,k)+t26(i,j,k))*rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1)))+
    9*(PQZ(i,j,1)/((0.860825768434464*PQZ(i,j,1)+1)*rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdlam*(hyu(i,j)*t47(i,j,1)-hyu(i-1,j)*t47(i-1,j,1)*PUZ(i-1,j,1)+
    rdth*(hxv(i,j)*t49(i,j,k-1)*PVZ(i,j,k-1)-hxv(i,j-1)*t49(i,j,k-2)*PVZ(i,j,k-2)))+
    Z(i,j,1)))+7.84806152880239E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    1,1)+rdth*((-t120(i,j,k)-t120(i,j,k+1)))+0.285714285714286*(PTZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+0.607824693422519*(PQZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+6.30915356075388E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    (0.860825768434464*PQZ(i,j,1)+1))/(hxt(i,j)*hyt(i,j))/(p(i,j,k+1)-p(i,j,k))
ELSE
  PDTDT(i,j,k) = (0.285714285714286*PTZ(i,j,k)*(7.84806152880239E-8*
    Z(i,j,1)+t18(i,j,k)+t26(i,j,k))*rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1)))+
    9*(PQZ(i,j,1)/((0.860825768434464*PQZ(i,j,1)+1)*rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdlam*(hyu(i,j)*t47(i,j,1)-hyu(i-1,j)*t47(i-1,j,1)*PUZ(i-1,j,1)+
    rdth*(hxv(i,j)*t49(i,j,k-1)*PVZ(i,j,k-1)-hxv(i,j-1)*t49(i,j,k-2)*PVZ(i,j,k-2)))+
    Z(i,j,1)))+7.84806152880239E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    1,1)+rdth*((-t120(i,j,k)-t120(i,j,k+1)))+0.285714285714286*(PTZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+0.607824693422519*(PQZ(i,j,k)/(rdth*(hxt(i,j)*t47(i,j,1)*PVZ(i,j,1)-hxt(i-1,j)*t47(i-1,j,1)*PVZ(i-1,j,1))))*rdth*((-t95(i,j,k)-t95(i,j,k+1)))+6.30915356075388E-8*(rdalb*((-t116(i,j,k)-t116(i,j,k+1)))+
    (0.860825768434464*PQZ(i,j,1)+1))/(hxt(i,j)*hyt(i,j))/(p(i,j,k+1)-p(i,j,k))
ENDIF
1140 CONTINUE
1130 CONTINUE
1120 CONTINUE

Research Seminar 2011
DYN Specification Size (LOC) Compared to Fortran 77, 95, and HPF
Performance Results

Cray T3D using shared memory

Cray T3D using Cray MPI
Related Work: Generating Numerical Algebra Libraries with FLAME

Algorithm: $[A] := LU_{BLK_{VAR1}}(A)$

Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Determine block size $b$
  Repartition

\[
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}
\]

where $A_{11}$ is $b \times b$

$A_{01} := U_{01} = L_{00}^{-1} A_{01}$
$A_{10} := L_{10} = A_{10} U_{00}^{-1}$
$A_{11} := LU(A_{11} - L_{10} U_{01})$

Continue with

\[
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}
\]

endwhile

FLAME library code
Example: Automatic Programming for XML Web Service Protocols

Started out in 1999 with the idea to enhance data interoperability with XML using compilers for static analysis and the use of code generators

Now in 2011 a $7.3M valued product “gSOAP” by Genivia Inc
Requirements and Benefits

• Requirements
  – High-level specification syntax and semantics, easy to write, easy to learn
  – Pure code at high levels: code that is side-effect free
  – Implicit parallelism, expressed at a high level

• Benefits
  – Saves time, assuming |spec| < |target code|
  – Increased software reliability
  – Simplified maintenance (re-generate code)
  – Enhanced portability, using platform-specific transformations
  – No low-level library API to learn
Automatic Programming in Industry

• Acceleo is an open source code generator for Eclipse, based on UML-like concepts
  – Model-Driven Architecture (MDA) approach
  – Meta-Object Facility (MOF) originates from UML
• Maple, Mathematica, Matlab support code generation
  – Design of programs by symbolic algebra from algorithm specs
  – Program codes are objects to manipulate
  – Matlab Simulink FPGA development
• Program “wizards” that allow programmers to design graphical user interfaces interactively
  – Mitigates the cost of GUI bloat by code template instantiations
• OOP, C++ templates, and generic programming
  – Code reuse, no algorithmic restructuring and thus inflexible
Questions?