Automatically Generating High-Performance Parallel Code for Atmospheric Simulation Models:

Challenges and Solutions for Auto-Programming Tools

Robert van Engelen
Acknowledgments

• **G. Cats**, Royal Netherlands Meteorological Institute, the Netherlands

• **Prof. L. Wolters**, high-performance computing group, Leiden University

• **Prof. W. Dewar**, Department of Oceanography, Florida State University

• **Dr. P. van der Mark**, Department of Scientific Computing, Florida State University
Overview

• Some comments on domain-specific modeling (DSM)
• Metamodeling using domain-specific languages (DSL):
  – When are there advantages compared to hand coding?
  – Code efficiency and machine targeting challenges?
• Experience designing and implementing a DSL for atmospheric models:
  – Weather forecast models (in this talk)
  – Coupled ocean-atmosphere models (not in this talk)
• Performance results
  – HIRLAM weather forecast system
• Conclusions
Domain-Specific Modeling

• Domain-specific modeling (DSM) is supported by specialized domain-specific languages (DSL)
• A DSL provides a *metamodeling* language

⇒ A DSL introduces well-defined notations (*syntax*) and concepts (*semantics*)
⇒ Code generation automates the creation of executable source code directly from the DSM
Model, Metamodel, and Meta-metamodel

Model (abstract)

Metamodel (DSL specification)

Generator (DSL translator)

Model code (for a machine)

Syntax defined by:
- common math notation
- set of domain-specific types
- template library

Semantics defined by:
- type systems
- term rewriting systems
- template instantiation
High level of abstraction: grids, fields, and PDEs

Low level of abstraction (machine-specific Fortran)
Battle Between Opposites

High level of abstraction
Ensures correctness of model
abstract model = metamodel

Low level of abstraction
Ensures efficiency of executable code

The semantic gap
DSL Metrics

- **Ease-of-use**: what is the DSL learning curve?
- **Expressiveness**: what are the DSL limitations?
- **Safety**: is the DSL type safe? Or can we express impossible constructs that cannot be mapped?
- **Efficiency**: how efficient is the translation? How efficient is the generated code?
- **Multi-platform**: does the DSL provide features to support high-performance platforms?
DSL Complexity

- A metamodel specification that uses standard notation for an abstract model can be very compact
  - Composition of standardized operations, e.g. matrix ops
- Suppose a metamodel specification is closer to a *programmatic* formulation of the model
- Then, what if

  $$\text{complexity(dslcode)} \approx \text{complexity(handcode)}$$

is it still useful to use a DSM approach?
Scientific Model Development and Machine Targeting Scenarios

<table>
<thead>
<tr>
<th>Single iteration</th>
<th>Multiple iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton scenario</td>
<td>Multi-platform scenario</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incremental scenario</th>
<th>Incremental multi-platform scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single platform</td>
<td>Multiple platforms</td>
</tr>
</tbody>
</table>

Number $M$ of development cycles

Number $N$ of target platforms
### Scientific Model Development and Machine Targeting Scenarios

<table>
<thead>
<tr>
<th>Incremental</th>
<th>Incremental multi-platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>• an experimental scientific model running on specialized hardware (Cray YMP, GPU)</td>
<td>• an experimental scientific model that runs on many machines</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Singleton</th>
<th>Multi-platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Numerical Recipes</td>
<td>• BLAS</td>
</tr>
<tr>
<td>• <code>&lt;math.h&gt;</code></td>
<td>• FFTW</td>
</tr>
<tr>
<td>• standard numerical libs</td>
<td>• FLAME</td>
</tr>
</tbody>
</table>

**Number $M$ of development cycles**

**Number $N$ of target platforms**
### Scientific Model Development and Machine Targeting Scenarios

<table>
<thead>
<tr>
<th>Number $M$ of development cycles</th>
<th>Single iteration</th>
<th>Multiple iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental</td>
<td>• Development cost saving ratio = $s \cdots sM$</td>
<td>Incremental multi-platform</td>
</tr>
<tr>
<td>• Development cost saving ratio = $s \approx 1$</td>
<td>• Development cost saving ratio = $sN \cdots sMN$</td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td>Single platform</td>
<td>Multi-platform</td>
</tr>
<tr>
<td>Multi-platform</td>
<td>Multiple platforms</td>
<td></td>
</tr>
</tbody>
</table>

Number $N$ of target platforms
HIRLAM Weather Forecast System

Measurements (update / 6 hr)

Analysis

Dynamics DYN + Helmholtz

Physics

Postprocessing & visualization

Forecast
DYN Specification Size (LOC) Compared to Fortran 77, 95, and HPF
Our Study on DYN for 3 Platforms
(Assuming LOC = Cost)

<table>
<thead>
<tr>
<th>Single iteration</th>
<th>Multiple iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Incremental</strong></td>
<td><strong>Incremental multi-platform</strong></td>
</tr>
<tr>
<td>LOC saving ratio = 2.9(M)</td>
<td>LOC saving ratio = 2.9(\times 3M) = 8.7(M)</td>
</tr>
<tr>
<td><strong>Singleton</strong></td>
<td><strong>Multi-platform</strong></td>
</tr>
<tr>
<td>LOC saving ratio (s = 2.9)</td>
<td>LOC saving ratio = 2.9(\times 3) = 8.7</td>
</tr>
</tbody>
</table>

- Single platform
- Multiple platforms (\(N = 3\))

Number \(N\) of target platforms

Number \(M\) of development cycles
A Formal, Systematic Approach

Type System
- Polymorphic (Milner-style inference)
- + Multi-type
- + Optimal type distance conversion

Term Rewriting System
- Derived from algebraic identities
- Typed code templates “codelets”
Polymorphic Type Systems

- Polymorphic type systems provide:
  - **Type safety** by type checking and inference
  - **Overloading**, so appropriate functions and operators can be picked automatically
  - **Type coercion** to automatically convert type-incompatible operands
Polymorphic Multi-Type Systems

• Multiple type systems (multi-type systems) utilize different polymorphic type categories
• We use three type systems:
  – Numerical types
    • Fields are float or complex, grid points are discrete integer
  – Dimension and unit types (Pressure, Distance, ...)
    • Dimensional analysis
    • Automatic scaling of constants
  – Arakawa grid types
    • Used for selecting centered finite difference operations, midpoint averaging, midpoint quadratures
Type Overloading and Coercion

• What to do when:
• Functions/operators are overloaded
• Actual operands and/or return types do not match any overloaded alternative?
  – For comparison: Ada, C++, ... compilers give up

```
int func(int, float)
float func(float, int)
bool func(bool, int)
```

```
x = (float) func(n, (float)(int) b)
x = func((float) n, (int) b)
x = (float) func((bool) n, (int) b)
...```

Making Polymorphism and Coercion
Part of the Translation Process

• The model uses abstract operators
• **Overloading**: select a matching overloaded operator as a concrete instantiation of the abstract operator (selection is visible in output)
• **Optimal type distance conversion**: if match fails then convert parameters and return value using *fewest total number of conversion operations*

```plaintext
\[ df(_, _) \text{`grid_overloaded} := [df_g, df_h, df_c].
\]
\[ df_g(_, :: \text{field x(grid)}, x) :: \text{field x(half)}.\]
\[ df_h(_, :: \text{field x(half)}, x) :: \text{field x(grid)}.\]
\[ df_c(_, :: \text{field x(grid)}, x) :: \text{field x(grid)}.\]
...
Arakawa Grid = Type System

\[ u :: \text{float dim Velocity field} \ (x(\text{half}), y(\text{grid}), z(\text{half}), t) \text{ on global.} \]
\[ v :: \text{float dim Velocity field} \ (x(\text{grid}), y(\text{grid}), z(\text{half}), t) \text{ on global.} \]
\[ p :: \text{float(0 .. 107000) dim Pressure field} \ (x(\text{grid}), y(\text{grid}), z(\text{grid}), t) \text{ monotonic k(+)} \text{ on global.} \]
\[ p_s :: \text{float(10000 .. 107000) dim Pressure field} \ (x(\text{grid}), y(\text{grid}), t) \text{ on horizontal.} \]
\[ p = A + B \times p_s. \]
\[ p_s_t :: \text{float dim Pressure/Time field} \ (x(\text{grid}), y(\text{grid}), z(\text{grid}), t) \text{ on horizontal.} \]
\[ d \ p_s \ / \ d \ t = - \text{integrate}(\text{nabla} \times [u, v], \text{eta} = \text{eta}(1) .. \text{eta(nlev+1))). \]
Dimensions and Units = Type System

\[ \frac{\partial p_s}{\partial t} = - \int_0^1 \nabla \cdot \mathbf{V} \, dz \quad (2) \]

\[ \mathbf{V} = \begin{pmatrix} u \\ v \end{pmatrix} \frac{\partial p}{\partial z} \quad (3) \]

\( V := [u_{aux}, v_{aux}] \).
\( V = [u, v] \times \frac{dp}{d\eta} \).
\( u_{aux} :: \text{float dim (Pressure * Velocity) field (x(half), y(grid), z(half), t) on global.} \)
\( v_{aux} :: \text{float dim (Pressure * Velocity) field (x(grid), y(half), z(half), t) on global.} \)
Programmable Term Rewrite System

Goal is to provide programmable untyped+typed TRS and to simplify the formulation of TRS rules for the user

Rewrite rules are of the form:

\[ \text{rewrite-class} \mid LHS \Rightarrow RHS \]

User can also specify a rewrite strategy:

\[ [ \text{rewrite-class1}, \text{rewrite-class2}, \ldots ] \]

Our TRS applies AC and exploits “aggregate operator commutativity” to match terms to LHS
A classic problem: using linearity of sum (directional):

\[
\begin{align*}
\text{lin} \mid \sum(X \cdot Y, I=L..U) & \Rightarrow X \cdot \sum(Y, I=L..U) \text{ if } FV(X, I) \\
\text{lin} \mid \sum(X, I=L..U) & \Rightarrow (U-L+1) \cdot X \text{ if } FV(X, I) \\
\text{lin} \mid \sum(I, I=L..U) & \Rightarrow \frac{(U^2+U-L^2+L)}{2}
\end{align*}
\]

Example reduction:

\[
\begin{align*}
\sum(\sum(2 \cdot j \cdot x[i], i=1..n), j=1..m) & \\
\Rightarrow \sum(2 \cdot j \cdot \sum(x[i], i=1..n), j=1..m) & \\
\Rightarrow 2 \cdot \sum(x[i], i=1..n) \cdot \sum(j, j=1..m) & \\
\Rightarrow (m^2 + m) \cdot \sum(x[i], i=1..n)
\end{align*}
\]
Operator Commutativity and AC

But this is not correct!
   at least not in many computational problems.

We can simply redefine the rules:

\[
\text{lin} \mid \sum(X \cdot Y, I=L..U) \Rightarrow X \cdot \sum(Y, I=L..U) \text{ if } FV(X, I)
\]

\[
\text{lin} \mid \sum(X, I=L..U) \Rightarrow (U-L+1) \cdot X \text{ if } FV(X, I)
\]

\[
\text{lin} \mid \sum(I, I=L..U) \Rightarrow (U^2+U-L^2+L)/2 \text{ if } FV(X, I)
\]

But this is not correct!

\[
\sum_{j=1}^{m} \sum_{i=1}^{n} 2j \cdot x_i \quad \rightarrow \quad \begin{cases} 
(m^2 + m) \sum_{i=1}^{n} x_i & \text{if } m > 0 \\
0 & \text{if } m \leq 0
\end{cases}
\]
Operator Commutativity and AC: Program Code Optimization

Loop fusion of two data-independent loops:

\[
fuse | \text{do}(X, I=I..U) \land \text{do}(Y, I=I..U) \Rightarrow \text{do}(X \land Y, I=I..U)
\]

```
DO i=1,n
  x[i] = b[j]
ENDDO
DO j=1,m
  DO i=1,n
    a[i] = a[i]*b[j]
  ENDDO
ENDDO
```

```
DO i=1,n
  x[i] = b[j]
ENDDO
DO j=1,m
  a[i] = a[i]*b[j]
ENDDO
ENDDO
```

\[
do(x[i]:=b[i], i=1..n) \land do(do(a[i]:=a[i]*b[j], i=1..n), j=1..m) \\
\Rightarrow do(x[i]:=b[i] \land do(a[i]:=a[i]*b[j], j=1..n), i=1..n)
\]
Applicability of Commutativity: Syntactic and Semantic Constraints

Special syntax for aggregate operators:

\[ F(\ldots, \ldots, i = lo..hi, \ldots, \ldots) \]

- local scope of \( i \)
- local \( i \) has a range
- applies here

Suppose \( F \) and \( G \) are allowed to commute, but they are dependent:

\[ F(\ G(\ldots, i=j..n), j=1..m) \]

Reason:
Commuting would change the variable scope and affect \( FV \), the set of free variables
Templates and Template Instantiation

Templates are *typed* rewrite rules:

\[
\text{reduce}(A :: T, D :: \text{domain}(\text{index}), F :: \text{associative}(T \to T \to T)) :: T \text{ function}
\]

\[
\{ \begin{align*}
& \text{reduce} := \text{unit}_{\text{element}}(F); \\
& \text{for } D \text{ do } \text{reduce} := \text{apply}_{\text{op}}(F, \text{reduce}, A)
\end{align*}
\}
\]

Example instantiation:

\[
\begin{align*}
\text{sum}(a[i], i=1..n) \\
\Rightarrow \text{reduce}(a[i], i=1..n, +) \\
\Rightarrow \{ r_1 := 0; \text{for } i = 1..n \text{ do } r_1 := r_1 + a[i] \} \\
\Rightarrow r_1 &= 0 \\
& \text{DO } j=1,n \\
& \quad r_1 = r_1 + a[i] \\
& \text{ENDDO}
\end{align*}
\]
Optimization

• Before template instantiation:
  – Algebraic simplification with the GPAS TRS
  – Common-subexpression elimination with DICE

• After template instantiation:
  – Loop optimization and restructuring with GPAS
  – Cleaning up of code
Why we Need Better CSE

• Traditional compiler common-subexpression elimination (CSE) is applied to scalars:

\[
\begin{align*}
x &:= a + b \\
y &:= a + b + c
\end{align*}
\]

\[
\begin{align*}
x &:= a + b \\
y &:= x + c
\end{align*}
\]
Why we Need Better CSE

• CSE is syntactic, no algebraic properties used:

\[
\begin{align*}
  x &:= a + b \\
  y &:= a + c + b
\end{align*}
\]

• CSE is scalar and non-array-based:

\[
\begin{align*}
  x[i] &:= a[i] - b[i] \\
  y[i] &:= a[i-1] - b[i-1]
\end{align*}
\]
DICE

- DICE: *Domain-Independent CSE*
- Applies CSE to arrays and uses some common algebraic properties (e.g. ring) to find matches

\[ expr_1 = \pm expr_2 \ T \]

where \( T \) is an index transformation

- For example:

\[
a[i,j,0] - b[i+1,j,0] = - ( b[i,j,k] - a[i-1,j,k] )[i+1,j,0] \]

\[ T \]
DICE

• For example, assume $f$ and $\Sigma$ commute:

\[ u_i = \sum_{j=1}^{n} f(v_{ij}) \]
\[ w_i = \sum_{j=1}^{n} v_{i,j} \]

\[ u_i = \sum_{j=1}^{n} f(v_{ij}) \]
\[ w_i = \sum_{j=1}^{n} v_{i,j} \]

\[ u[i] := \text{sum}(f(v[i,j]), j=1..n) \]
\[ w[i] := \text{sum}(v[i,j], j=1..n) \]

DICE recognizes iteration ranges of aggregate operators and local scope of the iteration variable to detect CSEs.
DICE

<table>
<thead>
<tr>
<th></th>
<th>unary</th>
<th>add/sub</th>
<th>mul/div</th>
<th>total</th>
<th>assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>No CSE</td>
<td>3</td>
<td>49</td>
<td>39</td>
<td>91</td>
<td>5</td>
</tr>
<tr>
<td>Pre-CSE</td>
<td>3</td>
<td>43</td>
<td>29</td>
<td>75</td>
<td>31</td>
</tr>
<tr>
<td>Post-CSE</td>
<td>3</td>
<td>43</td>
<td>34</td>
<td>80</td>
<td>17</td>
</tr>
<tr>
<td>DICE</td>
<td>3</td>
<td>38</td>
<td>22</td>
<td>63</td>
<td>20</td>
</tr>
</tbody>
</table>

Euler equations for an inviscid, compressible flow in a 2D geometry Arakawa E-grid

Pre-CSE: eliminate common sub-expressions from discrete equations
Post-CSE: eliminate common sub-expressions from Fortran code (compiler)
Platform-Dependent Code Generation

• Sequential

• Vector

• Dataparallel (Fortran 90 / HPF)

• Parallel subdomain splitting (MPI)
Subdomain Splitting

- Rows and columns overlap to compute finite differences
- Boundary values (Dirichlet) contain observed values
Shared Memory Multiprocessor

1. Fetch with overlapping regions (rows and columns)
2. Compute local region
3. Store results

Shared memory machines:
- Multicore CPUs
- Cray T3D (logically shared)
- Cell BE / GPU
Message Passing Multicomputer

1. Exchange overlapping regions (rows and columns)
2. Compute local region
3. Synchronize

Message passing machines:
• Cluster of workstations
• Cray T3D (PVM/MPI)
Parallelization Performance Results

Cray T3D using shared memory

Cray T3D using MPI
Vectorization Performance Results

![Bar chart showing performance results for different configurations of Fortran 77 and Cray C98 processors. The x-axis represents different configurations: Handcoded Fortran 77 w/ vec, Generated Fortran 77 w/ vec, and Generated Fortran 77 w/ vec loop opt. The y-axis represents time in milliseconds (ms). The chart compares the performance of Convex C4 and Cray C98 processors.]
Conclusions

• DSL design considerations are important
• Formal, systematic translation approach
  – Type systems
  – Term rewriting
    • Simplification, algebraic translation, code optimization
    • Typed template instantiation
• Implementation
  – We used the Prolog programming language
• Performance results show that the semantic gap can be bridged
In the Future...

Writing programs shall be akin to writing math, using the help of a computer (algebra) system to create and manipulate code
End
Cost Saving: Singleton Scenario

- One specification versus one hand-coded solution
- Specification makes only sense when:
  \[ \text{cost(dslcode)} < \text{cost(handcode)} \]

- Hence, development time savings (project speedup):
  \[ \text{saving ratio} = \frac{\text{cost(handcode)}}{\text{cost(dslcode)}} > 1 \]

- Difficult to obtain development time savings!
  - Unless dslcode specification uses common math/algebra standards extensively or handcode is very hard to write
Cost Saving: Multi-Platform Scenario

- $N$ redevelopments of the hand-coded solution are required, one for each platform
- Specification makes sense when:

\[
\text{cost(dslcode)} < N \times \text{cost(handcode)}
\]

- Hence, development time savings (project speedup):

\[
\text{saving ratio} = N \times \frac{\text{cost(handcode)}}{\text{cost(dslcode)}} \approx N
\]

- Assumes the specification/generator can be reused for each new platform
  - Examples in this category: BLAS, FFTW, FLAME
Cost Saving: Incremental Scenario

- $M$ incremental additions to the model over time
- Each addition requires substantial changes to the handcode
- Specification makes sense when:

\[
\text{cost}(\text{dslcode}) = \Sigma_i \text{cost}(\text{dslcode}_i) < M \times \text{cost}(\text{handcode})
\]

- Development time savings (project speedup):

\[
\text{saving ratio} = M \times \text{cost(\text{handcode})} / \text{cost(\text{dslcode})} \approx M
\]

- Assumes dslcode additions are orthogonal by compositionality of the DSL design
Cost Saving: Incremental Multi-Platform Scenario

- $M$ incremental additions to the model over time
- $N \times M$ redevelopments of the hand-coded solution are required, one for each addition and platform
- Specification makes sense when:

$$cost(dslcode) = \Sigma_i cost(dslcode_i) < N \times M \times cost(handcode)$$

- Development time savings (project speedup):

$$saving\ ratio = N \times M \times cost(handcode) / cost(dslcode) \approx N \times M$$

- Assumes $dslcode$ additions are orthogonal by compositionality of the DSL design
- Assumes the specification/generator can be reused to generate code for each new platform
Operator Commutativity and AC: Program Code Optimization

Loop invariant removal and statement parallelization:

\[
\text{inv } | \ do(X \& Y, I=L..U) \Rightarrow X \& do(Y, I=L..U) \text{ if } FV(X, I) \\
\text{inv } | \ do(X, I=L..U) \Rightarrow X \text{ if } FV(X, I) \\
\text{par } | \ X ; Y \Rightarrow X \& Y \text{ if indep}(X, Y)
\]

Example:

\[
do(do(a[i]:=a[i]*b[j]; x[i]:=b[i], i=1..n), j=1..m) \\
\Rightarrow do(x[i]:=b[i], i=1..n) \& do(do(a[i]:=a[i]*b[j], i=1..n), j=1..m)
\]

\[
\begin{align*}
\text{DO } j &= 1,m \\
\text{DO } i &= 1,n \\
a[i] &= a[i]*b[j] \\
x[i] &= b[j] \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{DO } i &= 1,n \\
x[i] &= b[j] \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } j &= 1,m \\
\text{DO } i &= 1,n \\
a[i] &= a[i]*b[j] \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
We will consider three types of codes:

- **dslcode**: specification of a computational model
- **gencode**: the autocode generator’s output of the model in a specific target programming language
- **handcode**: the hand-coded model in a specific target programming language
Scientific Model Development and Machine Targeting Scenarios

- **“Singleton”**: a standard numerical library implemented for one type of platform
- **“Multi-platform”**: a standard library code optimized for $N$ different types of (high-performance) platforms
- **“Incremental”**: a non-standard scientific model under (continuous) development for one specific type of platform
- **“Incremental multi-platform”**: a non-standard scientific model under development for $N$ different platforms