Overview of Parallelization Techniques I

Christian Lengauer

Fakultät für Informatik und Mathematik

16. June 2008, Schloss Rheinfels, St.Goar, Germany
Text-Based Loop Transformations:

- fission, fusion, permutation, skewing, unrolling, unswitching
  - easy to apply
  - in simple cases often all you need
  - do not support a search for the best solution
  - favor some solutions over others that may be better

Model-Based Loop Transformations:

- map source code to an execution model
- find the optimal parallel solution in this model
  - quality metric: a given objective function
  - search and transformation completely automatic
  - analysis and target code can become complex
  - optimality in the model need not imply efficient target code
The Basic Polytope Model

for $i = 1$ to $n$ do
    for $j = 0$ to $i + m$ do
        $A(i, j) = A(i-1, j) + A(i, j-1)$
    od
    $A(i, i+m+1) = A(i-1, i+m) + A(i, i+m)$
od

parfor $p = 1$ to $m+n$ do
    for $t = \max(p-1, 2*p-m-2)$ to $n+p-2$ do
        $A(2+t-p, p) = A(1+t-p, p) + A(2+t-p, p-1)$
    od
    if $p \geq m+1$ then
        $A(p-m, 1+p) = A(p-m, p) + A(p-m-1, p)$
    fi
od

source operation dependence graph  target operation dependence graph
Capabilities of the Basic Model

- Fully automatic dependence analysis
- Optimizing search of the best solution in the solution space, w.r.t. a given objective function
- Exemplary objective functions:
  - minimal number of parallel steps; minimal number of processors
  - minimal number of parallel steps; maximal throughput
  - minimal number of communications
- Challenge: efficient target code
- Standard example: square matrix product
  - source code:

```plaintext
for i = 0 to n - 1 do
    for j = 0 to n - 1 do
        for k = 0 to n - 1 do
            C(i, j) = C(i, j) + A(i, k) * B(k, j)
        od
    od
od
```
Example: Square Matrix Product

- Index space, dependences
- Parallel steps
- Square processor array
- Hexagonal processor array
Hexagonal Solution
Restrictions and Uses of the Basic Model

**Restrictions:**
- Loop bounds must be affine expressions in the outer loop indices and structure parameters.
- Array indices must be affine expressions in the loop indices.
- Assignments may be to array variables or scalar variables.
- Loop nests may be imperfect.
- Calls of subprograms are considered atomic, i.e., are not subject to parallelization.
- Pointer structures are not considered (unless coded as arrays).
- Target loop nests may be
  - synchronous (outer loops sequential),
  - asynchronous (outer loops parallel).

**Uses:**
- Loop parallelization
- Cache optimization
Extension 1: Conditional Statements in the Body

Consequence:

- Dependences vary between branches.

```plaintext
real A[0:2*N+1]
for i = 0 to N
    for j = 0 to N
        A[i+j+1] := ..
        if (P) then
            A[i+j] := ..
        end if
        .. := A[i+j]
    end for
end for
```
Extension 1: Conditional Statements in the Body

**Technique:**

- A precise reaching definition analysis that combines
  - the iterative solution of data flow equations,
  - discovers dependences between entire arrays,
  - can handle conditionals
- integer linear programming,
  - discovers dependences between individual array elements
- attaches conditions to dependences
- builds the unconditional union of all conditional dependences
Extension 2: WHILE Loops in the Loop Nest

Consequences:

- In WHILE dimensions, the number of loop steps is determined at run time.
- The static index space is not a polytope but a polyhedron.
- The dynamic index space in the unbounded direction is uneven (a “comb”).
Extension 2: Example (Convex Hull)

<table>
<thead>
<tr>
<th>n</th>
<th>node</th>
<th>nrsuc</th>
<th>suc</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>0</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>B, C, A, E, D</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>2</td>
<td>A, E</td>
<td>C, A, E, D</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>0</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>1</td>
<td>D</td>
<td>E, D</td>
</tr>
</tbody>
</table>
Extension 2: Example (Convex Hull)

\[ S_1: \text{for } n := 0 \text{ while } node[n] \neq \bot \text{ do} \]
\[ S_2: \quad rt[n, 0] := n \]
\[ S_3: \quad nxt[n] := 1 \]
\[ S_4: \text{for } d := 0 \text{ while } rt[n, d] \neq \bot \text{ do} \]
\[ S_5: \quad \text{if } \neg tag[n, rt[n, d]] \text{ then} \]
\[ S_6: \quad tag[n, rt[n, d]] := tt \]
\[ S_7: \quad \text{for } s := 0 \text{ to } nrsuc[rt[n, d]] - 1 \text{ do} \]
\[ S_8: \quad rt[n, nxt[n] + s] := suc[rt[n, d], s] \quad \text{enddo} \]
\[ S_9: \quad nxt[n] := nxt[n] + nrsuc[rt[n, d]] \quad \text{endif} \]
\[ \text{enddo} \]
\[ \text{endo} \]
\[ \text{endo} \]
Extension 2: Two Approaches

- **Conservative Approach:**
  - The control dependence of the WHILE loop is respected.
  - One WHILE loop remains sequential, but may be distributed.
  - A nest of WHILE loops may also be parallel.
  - Challenge: detecting the end of a “tooth” of the “comb”; solved for shared and distributed memory.

- **Speculative Approach:**
  - The control dependence of the WHILE loop is ignored.
  - One WHILE loop may be parallel.
  - Additional memory space may be required.
  - A rollback of iterations may be necessary.
  - Challenges: implementing rollback; avoiding rollback; minimizing memory consumption.
Extension 3: Index Set Splitting

**Idea:**
- Partition the index space automatically to break a dependence pattern and increase parallelism.

\[
\text{for } i = 0 \text{ to } 2 \times n - 1 \text{ do } \\
A(i, 0) = \ldots A(2 \times n - i - 1, 0) \\
\text{od}
\]

\[
\text{for } i = 0 \text{ to } n - 1 \text{ do } \\
A(i, 0) = \ldots A(2 \times n - i - 1, 0) \\
\text{od}
\]

**Technique:**
- Separate the sinks in the graph from the rest.
- Propagate splits backwards to the context
  - Challenge: termination in the presence of cycles (cut off)
  - Challenge: exponential growth in the number of sources (heuristics)
Extension 4: Tiling

- **Goal:** Determine the optimal granularity of parallelism
  - When? (Before or after the parallelization.)
  - How? (Shape, form and size of the tiles.)
  - What? (Space and/or time.)

- **When:** After the parallelization
  - It’s simpler and more widely applicable: only one perfect target loop nest.
  - It’s more powerful: flexible space-time mapping before inflexible tiling.

- **How:** Space and time separately
  - The risk: heuristics wins only with certain allocations.
  - The gain: precise and independent adaptation to hardware parameters.

- **What:**
  - Tiling space: adapts to resources (# processors).
  - Tiling time: adapts to performance (computation/communication ratio).
Goal: Avoid duplicate computations

Technique: Loop-carried code placement (LCCP)
- Identifies expressions that have the same value.
- Determines the optimal time and place for the evaluation.
- Determines the optimal place for the result.
- Hoists \( x \)-dimensional expressions out of \( y \)-dimensional loop nests \((y > x)\)

Example: Shallow Water Simulation

```fortran
FORALL (j=1:n, i=1:m) H(i, j) =
& P(i, j) + 0.25 * (U(i+1,j)*U(i+1,j) + U(i,j)*U(i,j)
& + V(i,j+1)*V(i,j+1) + V(i,j)*V(i,j))
```

↓

```fortran
FORALL (j=1:n, i=1:m+1) TMP1(i, j) = U(i,j)*U(i,j)
FORALL (j=1:n+1, i=1:m) TMP2(i, j) = V(i,j)*V(i,j)
FORALL (j=1:n, i=1:m) H(i, j) =
& P(i, j) + 0.25 * (TMP1(i+1,j) + TMP1(i,j)
& + TMP2(i,j+1) + TMP2(i,j))
```
Goal: Be able to handle array expressions of the form $A(p*i)$

"Parameter" $p$:
- Has a previously unknown but fixed value
- Typical case: Extent of the polytope in some fixed dimension

Application: Select a row or column of a matrix as a vector

Technique:
- Solve conflict equation system in $\mathbb{Z}$.
- Algorithm known for exactly one parameter.
- Math: entire quasi-polynomials.

Challenge: Dependence analysis
- Are the solutions inside or outside the iteration space? 
  (Solving the existence inequations...)
- What is the direction of a dependence? 
  (Establishing an order...)
Extension 7: Non-Affine Loop Bounds

**Goal:** Be able to scan domains with curved bounds.
- Curves must be described by polynomials.
- Domains are semi-algebraic sets
  (sets of solutions of inequation systems of polynomials in $\mathbb{Z}$).

**Applications:**
- Normal source code: Sieve of Eratosthenes (bound $i^2 \leq n$)
- Loops with non-constant stride:
  - Example:  
    ```
    for (j=0; j<=n; j+=i)
    → for (k=0; k*i<=n; k++)
    ```
  - in the loop body:  
    $$ j \rightarrow k*i $$
- Non-linear loop transformations:
  - Non-linear schedules can substantially improve the performance of
    solving affine recurrence equations (i.e., of executing loop nests) over
    linear schedules.

**Challenges:**
- Avoid non-affinities in the dependence analysis;
  postpone them to the code generation
- Code simplification.
for (x=1; x<=4; x++)
for (y=1; y<=9; y++)
    T1(x,y);
for (x=5; x<=7; x++) {
    for (y=1; y<=\[4-\sqrt{3x-12}\]; y++)
        T1(x,y);
    for (y=\[4+\sqrt{3x-12}\]; y<=9; y++)
        T1(x,y);
}
Extension 7: Cases and Techniques

- **Non-Linear Parameters**: e.g., $p^2i$, $pq^2i$, $pi$
  - LP solution methods like Fourier-Motzkin and Simplex can be generalized to handle several non-linear parameters.
  - Application: tiling and code generation.
  - Math: quantifier elimination in $\mathbb{R}$.

- **Also Non-Linear Variables**: e.g., $p^2i^2$, $p^2i^2$, $i^2j$
  - Math: Cylindrical algebraic decomposition.
  - Application: Code generation for scanning arbitrary semi-algebraic sets.
The Loop Parallelizer LooPo

- **Input:**
  - Loop code without parallelism (FORTRAN, C, recurrence equations)
  - Specification of a data flow graph (skip next step)

- **Dependence Analysis:** transition to the model
  - Method: Banerjee (restricted), Feautrier (complete), control flow fuzzy array dependence analysis (CfFADA, can also handle alternations)
  - Optional: index set splitting, single-assignment conversion

- **Space-Time Mapping:**
  - Schedule: Lamport (simple), Feautrier (complete), Darte-Vivien (compromise)
  - Allocation: Feautrier (complete), Dion-Robert (more practical), forward-communication only (prepares for tiling)

- **Code Generation:**
  - Based on the French loop code generator CLooG
  - Generates loops and communication
  - Tiles
Parallel Program Skeletons

Idea:
- Predefine frequently used patterns of parallel computation
- Specify each pattern as a higher-order function
- Provide implementations for a variety of parallel platforms
- Possibly use metaprogramming to make skeletons adaptive

Examples:
- Small scale: collective operations
  - data transfer: broadcast, scatter, gather, all-to-all
  - data transfer + computation: reduce, scan
- Larger scale: algorithmic patterns
  - divide-and-conquer, branch-and-bound
  - dynamic programming, searching in suffix trees
  - algorithms on labelled graphs

Technique:
- Functional source language: Template Haskell, MetaML, MetaOCaml
- Imperative target language: C, C+MPI,...
- Compilation step: no standard tools so far
**Collective Operations**

$t_s$: start-up time  \quad t_w$: per-word transfer time  \quad m$: blocksize

<table>
<thead>
<tr>
<th>composition rule</th>
<th>improvement if</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Scan}_1; \text{Reduce}_2 \rightarrow \text{Reduce}$</td>
<td>always</td>
</tr>
<tr>
<td>$\text{Scan}; \text{Reduce} \rightarrow \text{Reduce}$</td>
<td>$t_s &gt; m$</td>
</tr>
<tr>
<td>$\text{Scan}_1; \text{Scan}_2 \rightarrow \text{Scan}$</td>
<td>$t_s &gt; 2m$</td>
</tr>
<tr>
<td>$\text{Scan}; \text{Scan} \rightarrow \text{Scan}$</td>
<td>$t_s &gt; m(t_w + 4)$</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Scan} \rightarrow \text{Comcast}$</td>
<td>always</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Scan}_1; \text{Scan}_2 \rightarrow \text{Comcast}$</td>
<td>$t_s &gt; \frac{m}{2}$</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Scan}; \text{Scan} \rightarrow \text{Comcast}$</td>
<td>$t_s &gt; m(\frac{1}{2}t_w + 4)$</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Reduce} \rightarrow \text{Local}$</td>
<td>always</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Scan}_1; \text{Reduce}_2 \rightarrow \text{Local}$</td>
<td>always</td>
</tr>
<tr>
<td>$\text{Bcast}; \text{Scan}; \text{Reduce} \rightarrow \text{Local}$</td>
<td>$t_w + \frac{1}{m}t_s \geq \frac{1}{3}$</td>
</tr>
</tbody>
</table>
Divide-and-Conquer Hierarchy: Tasks

<table>
<thead>
<tr>
<th>skeleton</th>
<th>restriction</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcA</td>
<td>independent subproblems</td>
<td>Quicksort, maximum independent set</td>
</tr>
<tr>
<td>dcB</td>
<td>fixed recursion depth</td>
<td>$n$ queens</td>
</tr>
<tr>
<td>dcC</td>
<td>fixed division degree $k$</td>
<td>Karatsuba integer product ($k=3$)</td>
</tr>
</tbody>
</table>
Divide-and-Conquer Hierarchy: Data

<table>
<thead>
<tr>
<th>dcD</th>
<th>block recursion</th>
<th>triangular matrix inversion ($k=2$), Batcher sort ($k=2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcE</td>
<td>elementwise operations</td>
<td>matrix-vektor product ($k=4$)</td>
</tr>
<tr>
<td>dcF</td>
<td>communication between corresponding elements</td>
<td>Karatsuba polynomial product ($k=3$), bitonic merge ($k=2$), FFT ($k=2$), Strassen matrix product ($k=7$)</td>
</tr>
</tbody>
</table>
Skeleton Implementation

- **Principle:**
  - specification \( recX = \) iterative form \( itX \)
  - transition from Haskell to domain-specific language

- **dcA** (base, divide, combine, input)
  - dynamic allocation of time and space
  - no load balancing

- **dcF** (k, indeg, outdeg, basic, divide, combine, n, input)
  - static allocation of time and space via additional parameters
    - number of subproblems
    - division degree of input data
    - combination degree of output data
    - depth of recursion
  - dependence regular but not affine (no analysis necessary)
  - similar to the polytope model but no search for a schedule
  - symbolic size inference on skeleton parameters
**Skeleton Metaprogramming**

- **Metaprogramming:**
  - Using a meta language to transform programs in an object language
  - Both languages can be the same (*multi-stage programming*)

- **Advice:**
  - Use a functional metalanguage
  - Model the syntax for the object language by abstract data types
  - Exploit the type structure of the metalanguage for transformations

- **Adaptive Libraries:**
  - Old approach:
    - Add switch parameters to the library functions to customize
    - Can’t handle “new” cases without reprogramming
    - Caller can provide inconsistent information
  - New approach:
    - Perform an analysis on type and shape of the arguments
    - Provides consistency and flexibility
    - Can reduce abstraction penalty due to a lack of domain-specific knowledge considerably
Conclusions

- How do the methods perform?
  - Automation required high (affine) regularity.
  - Constant number of breaks in regularity can be handled.
  - Non-affinity requires sophisticated mathematics.
  - Code generation very difficult in general; heuristics help.

- Is it for ArtistDesign?
  - Loop parallelization probably only in special cases.
  - Skeletons have high potential – simple or sophisticated.
  - There is experience with tool prototypes.
  - Build dedicated tools.
References

Basic Polytope Model

Extension 1: Conditionals

Extension 2: WHILE Loops

Extension 3: Index Set Splitting
References

- **Extension 4: Tiling**


- **Extension 5: Expressions**

- **Extension 6: Non-Affine Index Array Expressions**

- **Extension 7: Non-Affine Loop Bounds**

Small-Scale Skeletons: Collective Operations

Large-Scale Skeletons: Divide-and-Conquer

Metaprogrammed Skeletons