Automatic Granularity-Aware Parallelization of Programs with Predicates, Functions, and Constraints

Manuel Hermenegildo\textsuperscript{1,2}

http://www.cliplab.org/~herme

with Francisco Bueno,\textsuperscript{1} Manuel Carro,\textsuperscript{1} Amadeo Casas,\textsuperscript{2}
Pedro López,\textsuperscript{1} Edison Mera,\textsuperscript{1} and Jorge Navas\textsuperscript{2}

Departments of Computer Science
\textsuperscript{1}Technical University of Madrid, and
\textsuperscript{2}University of New Mexico

Supported by several CICYT/MCyT/MEC grants and EU FP-4/5/6 projects.
Objectives

- Parallelism (*finally!*!) becoming mainstream thanks to *multicore* – even on laptops!
- Our objective herein is *automatic parallelization* of programs with predicates, functions, and constraints.
- We concentrate on detecting *and-parallelism* (corresponds to, e.g., loop parallelization, task parallelism, divide and conquer, etc.):
Objectives

- Parallelism (*finally!*!) becoming mainstream thanks to *multicore* – even on laptops!
- Our objective herein is *automatic parallelization* of programs with predicates, functions, and constraints.
- We concentrate on detecting *and-parallelism* (corresponds to, e.g., loop parallelization, task parallelism, divide and conquer, etc.):

```
fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-1)+fib(N-2)
   :- N>1.
```

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
   N>1,
   ( N1 is N-1,
     fib(N1, F1) ) &
   ( N2 is N-2,
     fib(N2, F2) ),
   F1+F2.
```

→ Need to detect *independent* tasks.
What is Independence? (for Functions, Predicates, Constraints, ...)

**Correctness:** “same” solutions as sequential execution.

**Efficiency:** execution time $< \text{than seq. program}$ (or, at least, no-slowdown: $\leq$).

(We assume parallel execution has no overhead in this first stage.)

Running $s_1 // s_2$:

<table>
<thead>
<tr>
<th></th>
<th>Imperative</th>
<th>Functions</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$Y := W+2$; $X := Y+Z$;</td>
<td>$(+W2)$; $(+Z)$</td>
<td>$Y = W+2$; $X = Y+Z$,</td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- read-write deps
- strictness
- cost!
What is Independence? (for Functions, Predicates, Constraints, ...)

- **Correctness**: “same” solutions as sequential execution.
- **Efficiency**: execution time \(<\) than seq. program (or, at least, *no-slowdown*: \(\leq\)).
  (We assume parallel execution has no overhead in this first stage.)

<table>
<thead>
<tr>
<th></th>
<th>Imperative</th>
<th>Functions</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_1)</td>
<td>(Y := W+2;)</td>
<td>(+ (W) 2)</td>
<td>(Y = W+2,)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(X := Y+Z;)</td>
<td>(+ (Z))</td>
<td>(X = Y+Z,)</td>
</tr>
<tr>
<td><strong>read-write deps</strong></td>
<td><strong>strictness</strong></td>
<td><strong>cost!</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>For <em>Predicates</em> (multiple procedure definitions):</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:-</td>
<td></td>
</tr>
<tr>
<td>(s_1) (p(X),)</td>
<td>(q(X) := X=a.)</td>
</tr>
<tr>
<td>(s_2) (q(X),)</td>
<td></td>
</tr>
<tr>
<td>(\text{write}(X).)</td>
<td>(q(X) := X=\text{b}, \text{large computation}.)</td>
</tr>
<tr>
<td></td>
<td>(q(X) := X=a.)</td>
</tr>
</tbody>
</table>

*Again, cost issue: if \(p\) affects \(q\) (prunes its choices) then \(q\) ahead of \(p\) is speculative.*

- **Independence**: condition that guarantees correctness *and efficiency.*
Independence

- **Strict independence** (suff. condition): no “pointers” shared at run-time:

- **Non-strict independence**: only one thread accesses each shared variable.
  - Requires global analysis.
  - Required in programs using “incomplete structures” (difference lists, etc.).
Independence

- **Strict independence** (suff. condition): no “pointers” shared at run-time:
  - Requires global analysis.
  - Required in programs using “incomplete structures” (difference lists, etc.).

- **Non-strict independence**: only one thread accesses each shared variable.
  - Requires global analysis.
  - Required in programs using “incomplete structures” (difference lists, etc.).

- **Constraint independence** – more involved:
  
  \[
  \text{main} :- X \succ Y, Z \succ Y, p(X) \& q(Z), \ldots
  \]
  \[
  \text{main} :- X \succ Y, Y \succ Z, p(X) \& q(Z), \ldots
  \]
**Independence**

- **Strict independence** (suff. condition): no "pointers" shared at run-time:
  - Requires global analysis.
  - Required in programs using “incomplete structures” (difference lists, etc.).

- **Non-strict independence**: only one thread accesses each shared variable.
  - Requires global analysis.
  - Required in programs using “incomplete structures” (difference lists, etc.).

- **Constraint independence**: more involved:
  
  \[
  \text{main} :- X .> Y, Z .> Y, \ p(X) \ & q(Z), \ldots \\
  \text{main} :- X .> Y, Y .> Z, \ p(X) \ & q(Z), \ldots
  \]

  **Sufficient a-priori condition**: given \( g_1(x) \) and \( g_2(y) \), c state just before them:
  
  \[
  (x \cap y \subseteq \text{def}(c)) \text{ and } (\exists x \ c \& \exists y \ c \rightarrow \exists y \cup x \ c)
  \]

  \( (\text{def}(c) = \text{set of variables constrained to a unique value in } c) \)

  - For \( c = \{x > y, z > y\} \) \quad \exists \{x\} c = \exists \{z\} c = \exists \{x, z\} c = \text{true}
  
  - For \( c = \{x > y, y > z\} \) \quad \exists \{x\} c = \exists \{z\} c = \text{true}, \quad \exists \{x, z\} c = x > z

  **Approximation**: presence of “links” through the store.
Parallelization Process

- Conditional dependency graph (of some code segment, e.g., a clause):
  - Vertices: possible tasks (statements, calls,...),
  - Edges: possible dependencies (labels: conditions needed for independence).
- Local or global analysis used to reduce/remove checks in the edges.
- Annotation process converts graph back to parallel expressions in source.

```prolog
foo(\ldots) :-
g1(\ldots),
g2(\ldots),
g3(\ldots).
```

Local/Global analysis and simplification

( test(1–3) ⊨ ( g1, g2 ) & g3 ; g1, ( g2 & g3 ) )

Alternative: `g1, ( g2 & g3 )`
Concrete System Used in Examples: Ciao

- One of the popular Prolog/CLP systems (supports ISO-Prolog fully).
- At the same time, new-generation *multi-paradigm* language/prog.env. with:
  - Predicates, constraints, functions (including lazyness), higher-order, ...
    (And Prolog impure features only present as compatibility libraries.)
Concrete System Used in Examples: **Ciao**

- One of the popular Prolog/CLP systems (supports ISO-Prolog fully).
- At the same time, new-generation *multi-paradigm* language/prog.env. with:
  - Predicates, constraints, functions (including lazyness), higher-order, ...
    (And Prolog impure features only present as compatibility libraries.)
  - Assertion language for expressing rich program properties
    (types, shapes, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, ...).
  - Static debugging, verification, program certification, PCC, ...
Concrete System Used in Examples: Ciao

- One of the popular Prolog/CLP systems (supports ISO-Prolog fully).
- At the same time, new-generation *multi-paradigm* language/prog.env. with:
  - Predicates, constraints, functions (including lazyness), higher-order, ...
    (And Prolog impure features only present as compatibility libraries.)
  - Assertion language for expressing rich program properties
    (types, shapes, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, ...).
  - Static debugging, verification, program certification, PCC, ...
- Parallel, concurrent, and distributed execution primitives.
  - Automatic parallelization.
  - Automatic granularity and resource control.
Concrete System Used in Examples: Ciao

- One of the popular Prolog/CLP systems (supports ISO-Prolog fully).
- At the same time, new-generation multi-paradigm language/prog.env. with:
  - Predicates, constraints, functions (including lazyness), higher-order, ...
    (And Prolog impure features only present as compatibility libraries.)
  - Assertion language for expressing rich program properties
    (types, shapes, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, ...).
  - Static debugging, verification, program certification, PCC, ...
- Parallel, concurrent, and distributed execution primitives.
  - Automatic parallelization.
  - Automatic granularity and resource control.
- + several control rules (e.g., bf, id, Andorra), objects, syntactic/semantic extensibility, LGPL, ...
Some Speedups (for different analysis abstract domains)

Benchmark: ann

The parallelizer, self-parallelized
Granularity Control

- Replace parallel with sequential execution based on task size and overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate at compile time, even w/abstract interpretation).

```prolog
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...```

DAMP’07 – Nice, France – January 16, 2007
Granularity Control

- Replace parallel with sequential execution based on task size and overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate at compile time, even w/abstract interpretation).

```prolog
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ... 
```

Our approach:
- Derive at compile-time cost *functions* (to be evaluated at run-time) that efficiently bound task size (lower, upper *bounds*).
- Transform programs to carry out run-time granularity control.

For `inc_all`, (assuming “threshold” is 100 units):

```prolog
main :- read(X), read(Z), ( 2*length(X)+1 > 100 -> inc_all(X,Y) & r(Z,M) ; inc_all(X,Y), r(Z,M) ), ... 
```
Inference of Bounds on Argument Sizes and Procedure Cost in CiaoPP

1. Perform type/mode inference:
   $$\text{:- true inc\_all}(X,Y) : \text{list}(X,\text{int}), \text{var}(Y) \Rightarrow \text{list}(Y,\text{int}).$$

2. Infer size measures: list length.

3. Use data dependency graphs to determine the relative sizes of structures that variables point to at different program points – infer argument size relations:
   - \(\text{Size}_{\text{inc\_all}}^2(0) = 0\) (boundary condition from base case),
   - \(\text{Size}_{\text{inc\_all}}^2(n) = 1 + \text{Size}_{\text{inc\_all}}^2(n - 1)\).
   - \(\text{Sol} = \text{Size}_{\text{inc\_all}}^2(n) = n\).

4. Use this, set up recurrence equations for the computational cost of procedures:
   - \(\text{Cost}_{\text{inc\_all}}^L(0) = 1\) (boundary condition from base case),
   - \(\text{Cost}_{\text{inc\_all}}^L(n) = 2 + \text{Cost}_{\text{inc\_all}}^L(n - 1)\).
   - \(\text{Sol} = \text{Cost}_{\text{inc\_all}}^L(n) = 2n + 1\).

We obtain lower/upper bounds on task granularities.

Non-failure (absence of exceptions) analysis needed for lower bounds.
**Refinements (1): Granularity Control Optimizations**

- **Simplification of cost functions:**

  \[\ldots, (\text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \land r(Z,M)) \]
  \[\quad; \quad \text{inc\_all}(X,Y), r(Z,M)) , \ldots\]
Refinements (1): Granularity Control Optimizations

Simplification of cost functions:

..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M) 
  ; inc_all(X,Y), r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M) 
  ; inc_all(X,Y), r(Z,M) ), ...
Refinements (1): Granularity Control Optimizations

- Simplification of cost functions:
  
  ..., (length(X) > 50 → inc_all(X,Y) & r(Z,M)
  ; inc_all(X,Y), r(Z,M)), ...

  ..., (length_gt(LX,50) → inc_all(X,Y) & r(Z,M)
  ; inc_all(X,Y), r(Z,M)), ...

- Complex thresholds: use also communication cost functions, load, ...

[Example:] Assume \(\text{CommCost}(\text{inc\_all}(X)) = 0.1 \ (\text{length}(X) + \text{length}(Y))\).

We know \(\text{ub\_length}(Y)\) (actually, exact size) = \(\text{length}(X)\); thus:

\[
2 \ \text{length}(X) + 1 > 0.1 \ (\text{length}(X) + \text{length}(X)) \ \cong \\
2 \ \text{length}(X) > 0.2 \ \text{length}(X) \ \cong \\
2 > 0.2
\]
Refinements (1): Granularity Control Optimizations

- Simplification of cost functions:
  
  ..., (length(X) > 50 -> inc_all(X,Y) & r(Z,M)
  ; inc_all(X,Y), r(Z,M)), ...

  ..., (length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
  ; inc_all(X,Y), r(Z,M)), ...

- Complex thresholds: use also communication cost functions, load, ...

  **Example:** Assume $CommCost(inc_all(X)) = 0.1 \times (length(X) + length(Y))$. We know $ub\_length(Y)$ (actually, exact size) = $length(X)$; thus:

  \[
  2 \times length(X) + 1 > 0.1 \times (length(X) + length(X)) \iff
  2 \times length(X) > 0.2 \times length(X) \equiv
  \]

  Guaranteed speedup for any data size! \[\iff\] \(2 > 0.2\)
Simpilication of cost functions:

..., (length(X) > 50 -> inc_all(X,Y) & r(Z,M)
    ; inc_all(X,Y) , r(Z,M) ), ...

..., (length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
    ; inc_all(X,Y) , r(Z,M) ), ...

Complex thresholds: use also communication cost functions, load, ...

Example: Assume \(\text{CommCost}(\text{inc\_all}(X)) = 0.1 (\text{length}(X) + \text{length}(Y)).\)

We know \(\text{ub\_length}(Y)\) (actually, exact size) = \(\text{length}(X)\); thus:

\[
2 \text{length}(X) + 1 > 0.1 (\text{length}(X) + \text{length}(X)) \equiv \\
2 \text{length}(X) > 0.2 \text{length}(X) \equiv
\]

Guaranteed speedup for any data size! \(\leftarrow\) \(2 > 0.2\)

Checking of data sizes can be stopped once under threshold.

Data size computations can often be done on-the-fly.

Static task clustering (loop unrolling), static placement, etc.
Granularity Control System Output Example

\[
g_{\text{qsort}}([], []). \\
g_{\text{qsort}}([\text{First}|L1], L2) :- \\
\quad \text{partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg)}, \\
\quad \text{Size_Ls} > 20 \rightarrow (\text{Size_Lg} > 20 \rightarrow g_{\text{qsort}}(Ls, Ls2) \& g_{\text{qsort}}(Lg, Lg2) \\
\qquad ; g_{\text{qsort}}(Ls, Ls2), s_{\text{qsort}}(Lg, Lg2)) \\
\quad ; (\text{Size_Lg} > 20 \rightarrow s_{\text{qsort}}(Ls, Ls2), g_{\text{qsort}}(Lg, Lg2) \\
\qquad ; s_{\text{qsort}}(Ls, Ls2), s_{\text{qsort}}(Lg, Lg2))), \\
\quad \text{append}(Ls2, [\text{First}|Lg2], L2).
\]

\[
\text{partition3o4o}(F, [], [], [], 0, 0). \\
\text{partition3o4o}(F, [X|Y], [X|Y1], Y2, SL, SG) :- \\
\quad X =< F, \text{partition3o4o}(F, Y, Y1, Y2, SL1, SG), SL \text{ is } SL1 + 1. \\
\text{partition3o4o}(F, [X|Y], Y1, [X|Y2], SL, SG) :- \\
\quad X > F, \text{partition3o4o}(F, Y, Y1, Y2, SL, SG1), SG \text{ is } SG1 + 1.
\]
Refinements (2): Granularity-Aware Annotation

With classic annotators (MEL, UDG, CDG, ... ) we applied granularity control after parallelization:

\[ g_1 \rightarrow g_3 \]
\[ g_2 \rightarrow \text{"Annotation"} \rightarrow g_1, (g_2 \& g_3 ) \]

Granularity Control:
\[ g_1, (\text{gran\_cond} \rightarrow g_2 \& g_3 ; g_2, g_3 ) \]
Refinements (2): Granularity-Aware Annotation

With classic annotators (MEL, UDG, CDG, . . . ) we applied granularity control after parallelization:

Developed new annotation algorithm that takes task granularity into account:

- Annotation is a heuristic process (several alternatives possible).
- Taking task granularity into account during annotation can help make better choices and speed up annotation process.
- Tasks with larger cost bounds given priority, small ones not parallelized.
Granularity-Aware Annotation: Concrete Example

Consider the clause: \[ p : - a, b, c, d, e. \]

Assume that the dependencies detected between the subgoals of \( p \) are given by:

\[ \begin{align*}
& a \rightarrow b \\
& c \rightarrow d \\
& d \rightarrow e
\end{align*} \]

Assume also that:

\[ T(a) < T(c) < T(e) < T(b) < T(d), \]

where \( T(i) < T(j) \) means: cost of subgoal \( i \) is smaller than the cost of \( j \).
Granularity-Aware Annotation: Concrete Example

Consider the clause: \( p :\ :- a, b, c, d, e.\)

Assume that the dependencies detected between the subgoals of \( p \) are given by:

\[
\text{MEL annotator: } (a, b & c, d & e) \\
\text{UDG annotator: } (c & (a, b, e), d) \\
\text{Granularity-aware: } (a, c, (b & d), e)
\]

Assume also that:

\[ T(a) < T(c) < T(e) < T(b) < T(d), \]

where \( T(i) < T(j) \) means: cost of subgoal \( i \) is smaller than the cost of \( j \).
Refinements (3): Using Execution Time Bounds/Estimates

- Use estimations/bounds on *execution time* for controlling granularity (instead of steps/reductions).

- Execution time generally dependent on platform characteristics ($\approx$ constants) and input data sizes (unknowns).

- Platform-dependent, one-time calibration using fixed set of programs:
  - Obtains value of the platform-dependent constants (costs of basic operations).

- Platform-independent, compile-time analysis:
  - Infers cost functions (using modification of previous method), which return count of *basic operations* given input data sizes.
  - Incorporate the constants from the calibration.

  → we obtain functions yielding *execution times* depending on size of input.

- Predicts execution times with *reasonable* accuracy (challenging!).

- Improving by taking into account lower level factors (current work).
Consider \texttt{nrev} with mode:
\[ :\text{- pred nrev/2 : list(int) } * \text{ var.} \]

Estimation of execution time for a concrete input—consider:
\[ A = [1,2,3,4,5], \quad \overline{n} = \text{length}(A) = 5 \]

<table>
<thead>
<tr>
<th>component</th>
<th>Once ( K_{\omega_i} )</th>
<th>Static Analysis ( \text{Cost}_p(I(\omega_i), \overline{n}) = C_i(\overline{n}) )</th>
<th>Application ( C_i(5) \times K_{\omega_i} \times C_i(5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>21.27</td>
<td>( 0.5 \times n^2 + 1.5 \times n + 1 )</td>
<td>21 \times 446.7</td>
</tr>
<tr>
<td>nargs</td>
<td>9.96</td>
<td>( 1.5 \times n^2 + 3.5 \times n + 2 )</td>
<td>57 \times 567.7</td>
</tr>
<tr>
<td>giunif</td>
<td>10.30</td>
<td>( 0.5 \times n^2 + 3.5 \times n + 1 )</td>
<td>31 \times 319.3</td>
</tr>
<tr>
<td>gounif</td>
<td>8.23</td>
<td>( 0.5 \times n^2 + 0.5 \times n + 1 )</td>
<td>16 \times 131.7</td>
</tr>
<tr>
<td>viunif</td>
<td>6.46</td>
<td>( 1.5 \times n^2 + 1.5 \times n + 1 )</td>
<td>45 \times 290.7</td>
</tr>
<tr>
<td>vounif</td>
<td>5.69</td>
<td>( n^2 + n )</td>
<td>30 \times 170.7</td>
</tr>
</tbody>
</table>

Execution time \( \overline{K}_\Omega \times \text{Cost}_p(I(\Omega), \overline{n}) \): 1926.8
Visualization of And-parallelism - (small) qsort, 4 processors
Fib 15, 1 processor
Fib 15, 8 processors (same scale)
Fib 15, 8 processors (full scale)
Fib 15, 8 processors, with granularity control (same scale)