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

Manuel Hermenegildo$^{1,2}$

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

with Francisco Bueno$^1$, Manuel Carro$^1$, Amadeo Casas$^2$, Pedro López$^1$, Edison Mera$^1$, and Jorge Navas$^2$

Departments of Computer Science

$^1$Technical University of Madrid, and
$^2$University of New Mexico
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.):

\[
\begin{align*}
\text{fib}(0) & : = 0. \\
\text{fib}(1) & : = 1. \\
\text{fib}(N) & : = \text{fib}(N-1)+\text{fib}(N-2) : \text{N}>1.
\end{align*}
\]

\[
\begin{align*}
\text{fib}(0, 0). \\
\text{fib}(1, 1). \\
\text{fib}(N, F) : - \\
\quad \text{N}>1, \\
\quad ( \text{N1 is N-1,} \\
\quad \quad \text{fib(N1, F1)) & \&} \\
\quad ( \text{N2 is N-2,} \\
\quad \quad \text{fib(N2, F2))}, \\
\quad \quad \text{F1+F2}.
\end{align*}
\]

→ Need to detect *independent* tasks.
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></td>
<td>read-write deps</td>
<td>strictness</td>
<td>cost!</td>
</tr>
</tbody>
</table>

- **Running** \( s_1 // s_2 \):
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>read-write deps</td>
<td>strictness</td>
<td>cost!</td>
<td></td>
</tr>
</tbody>
</table>

- Running $s_1 // s_2$:

<table>
<thead>
<tr>
<th>main:</th>
<th>p(X), $s_1$, q(X), $s_2$, write(X).</th>
<th>p(X) :- X=a.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>q(X) :- X=b, large computation.</td>
</tr>
<tr>
<td></td>
<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:
- 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:
  
  main :- X >. Y, Z >. Y, p(X) & q(Z), ...
  main :- X >. Y, Y >. Z, p(X) & q(Z), ...
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.).
- Constraint independence – more involved:
  
  \[
  \text{main :- X .>. Y, Z .>. Y, p(X) & q(Z), ...} \\
  \text{main :- X .>. Y, Y .>. Z, p(X) & q(Z), ...}
  \]

  Sufficient a-priori condition: given \(g_1(x)\) and \(g_2(y)\), \(c\) state just before them:

  \[
  \begin{align*}
  & (\bar{x} \cap \bar{y} \subseteq \text{def}(c)) \text{ and } (\exists_{\bar{x}c} \land \exists_{\bar{y}c} \rightarrow \exists_{\bar{y} \cup \bar{x}c})
  
  \end{align*}
  \]

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

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

  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(...) :-
g1(...),
g2(...),
g3(...).
```

```
Local/Global analysis and simplification
```

```
( test(1–3) -> ( g1, g2 ) & g3 ; g1, ( g2 & g3 ) )
```

**Alternative:**

```
g1, ( g2 & g3 )
```

**Diagram:**

- Vertices: g1, g2, g3
- Edges: icond(1–3), icond(1–2), icond(2–3)
- Annotation process converts graph back to parallel expressions in source.
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

Number of Processors

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

P*S
P*SF/SF
P
S
L
N

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), ...
```
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:

\[
\text{:- true inc_all}(X,Y) : \text{list}(X,\text{int}), \text{var}(Y) => \text{list}(Y,\text{int}).
\]

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:

\[
\begin{align*}
\text{Size}_{\text{inc_all}}(0) &= 0 \text{ (boundary condition from base case)}, \\
\text{Size}_{\text{inc_all}}(n) &= 1 + \text{Size}_{\text{inc_all}}(n - 1).
\end{align*}
\]

\[\text{Sol} = \text{Size}_{\text{inc_all}}(n) = n.\]

4. Use this, set up recurrence equations for the computational cost of procedures:

\[
\begin{align*}
\text{Cost}_{\text{inc_all}}^L(0) &= 1 \text{ (boundary condition from base case)}, \\
\text{Cost}_{\text{inc_all}}^L(n) &= 2 + \text{Cost}_{\text{inc_all}}^L(n - 1).
\end{align*}
\]

\[\text{Sol} = \text{Cost}_{\text{inc_all}}^L(n) = 2n + 1.\]

- 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:
  
  ..., ( length(X) > 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) ), ...

Refinements (1): Granularity Control Optimizations

- Simplification of cost functions:
  
  \[
  \ldots, (\text{length}(X) > 50 \rightarrow \text{inc}_\text{all}(X,Y) \land r(Z,M) \\
  \quad \land \text{inc}_\text{all}(X,Y), r(Z,M)), \ldots
  \]

  \[
  \ldots, (\text{length} \_\text{gt}(LX,50) \rightarrow \text{inc}_\text{all}(X,Y) \land r(Z,M) \\
  \quad \land \text{inc}_\text{all}(X,Y), r(Z,M)), \ldots
  \]

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

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

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

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

- Simplification of cost functions:
  
  $$\ldots, (\text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \ &\ r(Z,M)\ ;\ \text{inc\_all}(X,Y),\ r(Z,M)), \ldots$$

  $$\ldots, (\text{length}\_gt(LX,50) \rightarrow \text{inc\_all}(X,Y) \ &\ r(Z,M)\ ;\ \text{inc\_all}(X,Y),\ r(Z,M)), \ldots$$

- 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:

  $$2 \text{length}(X) + 1 > 0.1 (\text{length}(X) + \text{length}(X)) \Leftrightarrow$$
  $$2 \text{length}(X) > 0.2 \text{length}(X) \equiv$$

  Guaranteed speedup for any data size! $\Leftrightarrow 2 > 0.2$
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 \( \text{CommCost}(\text{inc}_\text{all}(X)) = 0.1 \ (\text{length}(X) + \text{length}(Y)). \)
  
  We know \( \text{ub}_\text{length}(Y) \) (actually, exact size) = \( \text{length}(X) \); thus:

  \[
  2 \text{length}(X) + 1 > 0.1 \ (\text{length}(X) + \text{length}(X)) \Leftrightarrow \\
  2 \text{length}(X) > 0.2 \text{length}(X) \equiv \\
  \text{Guaranteed speedup for any data size!} \quad \Leftrightarrow \quad 2 > 0.2
  \]

- 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

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

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

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

\[ g_1, (g_2 \land g_3) \quad \text{Gran. Control} \quad g_1, (\text{gran}_\text{cond} \rightarrow g_2 \land g_3 ; g_2, g_3) \]
Refinements (2): Granularity-Aware Annotation

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

  \[ g_1 \xrightarrow{\text{test}(1-3)} g_3 \]
  \[ g_2 \]
  
  "Annotation" \quad g_1, \ (g_2 \& g_3) \quad \text{Gran. Control} \quad g_1, \ (\text{gran}_\text{cond} \rightarrow g_2 \& g_3 \ ; \ g_2, g_3)

- 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.

  \[ g_1 \xrightarrow{\text{test}(1-3)} g_3 \]
  \[ g_2 \]
  
  \text{Granularity-driven annotation} \quad (\text{gran}_\text{cond}, \text{test}_{13} \rightarrow (g_1, g_2) \& g_3 \ ; \ g_1, g_2, g_3)
  
  (assuming g_2 "small" and g_1 large if gran_cond)
Granularity-Aware Annotation: Concrete Example

- Consider the clause:  \( p :\neg a, b, c, d, e. \)
- Assume that the dependencies detected between the subgoals of \( p \) are given by:

\[
\begin{array}{c}
\text{Assume also that:} \\
T(a) < T(c) < T(e) < T(b) < T(d),
\end{array}
\]

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: \[ p :\leftarrow a, b, c, d, e. \]

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

\[
\begin{array}{c}
\text{MEL annotator:} \\
\text{UDG annotator:} \\
\text{Granularity-aware:}
\end{array}
\]

- 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.
    \[\rightarrow\] 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).
Execution Time Estimation: Concrete Example

- Consider \texttt{nrev} with mode:
  \[
  \texttt{:- pred nrev/2 : list(int) * var.}
  \]

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

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

Execution time $\overline{K_\Omega} \cdot \text{Cost}_p(I(\Omega), \bar{n})$: 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)