# 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.):

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

→ 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.)
- Running $s_1$ // $s_2$:

| | *Imperative* | *Functions* | *Constraints* |
|---|---|---|---|
| $s_1$ | `Y := W+2;` | `(+ W 2)` | `Y = W+2,` |
| $s_2$ | `X := Y+Z;` | `(+        Z)` | `X = Y+Z,` |
| | *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.)
- Running $s_1$ // $s_2$:

| | *Imperative* | *Functions* | *Constraints* |
|---|---|---|---|
| $s_1$ | `Y := W+2;` | `(+ W 2)` | `Y = W+2,` |
| $s_2$ | `X := Y+Z;` | `(+ Z)` | `X = Y+Z,` |
| | *read-write deps* | *strictness* | *cost!* |

| For *Predicates* (multiple procedure definitions): | |
|---|---|
| `main :-`<br>$s_1$ `p(X),`<br>$s_2$ `q(X),`<br>`write(X).` | `p(X) :- X=a.`<br>---<br>`q(X) :- X=b,` *large computation*.<br>`q(X) :- X=a.` |
| *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 independennce –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 independennce –more involved:

```
main :- X .>. Y, Z .>. Y, p(X) & q(Z), ...
main :- X .>. Y, Y .>. Z, p(X) & q(Z), ...
```

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

$$\boxed{(\bar{x} \cap \bar{y} \subseteq def(c))\ and\ (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow\ \exists_{-\bar{y} \cup \bar{x}} c)}$$

($def(c) =$ set of variables constrained to a unique value in $c$)

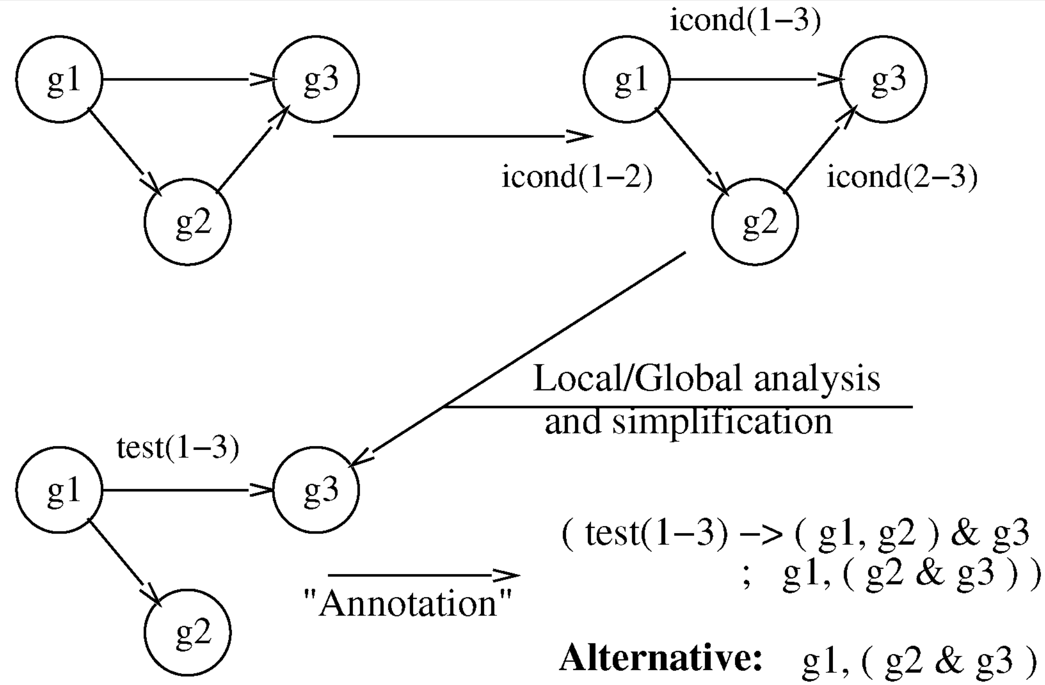
- For $c = \{x > y, z > y\}$ $\quad \bar{\exists}_{-\{x\}} c = \bar{\exists}_{-\{z\}} c = \bar{\exists}_{-\{x,z\}} c = true$
- For $c = \{x > y, y > z\}$ $\quad \bar{\exists}_{-\{x\}} c = \bar{\exists}_{-\{z\}} c = true, \quad \bar{\exists}_{\{x,z\}} c = x > z$

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.

```
foo(...) :-
     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, ...

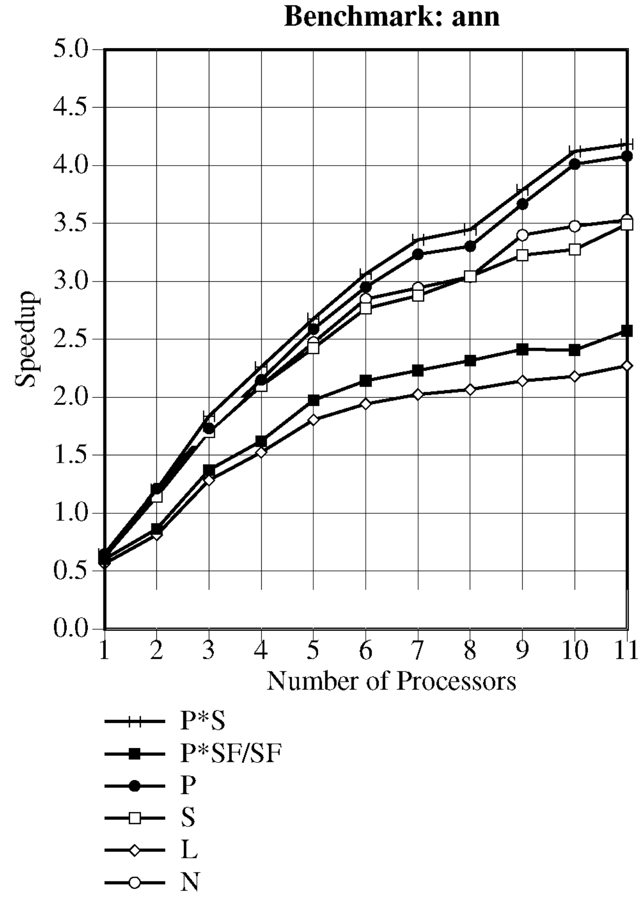
# Concrete System Used in Examples: `Ciao`

- 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)




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).

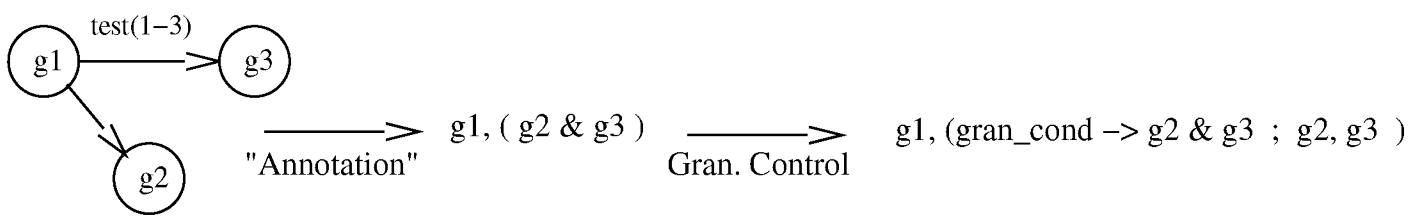
```
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...
```

# 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).

```
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):

```
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:

```
:- true inc_all(X,Y) : list(X,int), var(Y) => list(Y,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:

   $\mathtt{Size}^2_{\mathtt{inc\_all}}(0) = 0$ (boundary condition from base case),
   $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Size}^2_{\mathtt{inc\_all}}(n-1)$.

   Sol = $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = n$.

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

   $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(0) = 1$ (boundary condition from base case),
   $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n-1)$.

   Sol = $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2\,n + 1$.

- 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:

```
..., ( 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 $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$.
  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$
$$2 > 0.2$$

# 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 $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$.
  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

  Guaranteed speedup for any data size! $\Leftarrow$ $2 > 0.2$

# 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 $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$.
  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$

  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.

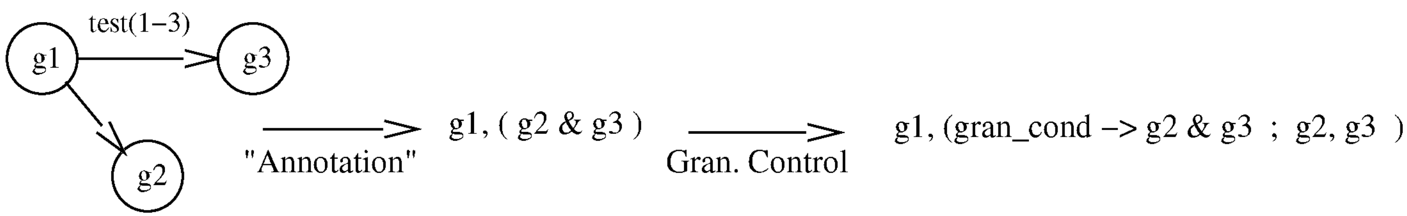
# Granularity Control System Output Example

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
  Size_Ls > 20 -> (Size_Lg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2)
                                 ;  g_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))
               ;  (Size_Lg > 20 -> s_qsort(Ls, Ls2) , g_qsort(Lg, Lg2)
                                 ;  s_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))),
  append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
  X =< F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
   X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), SG is SG1 + 1.
```

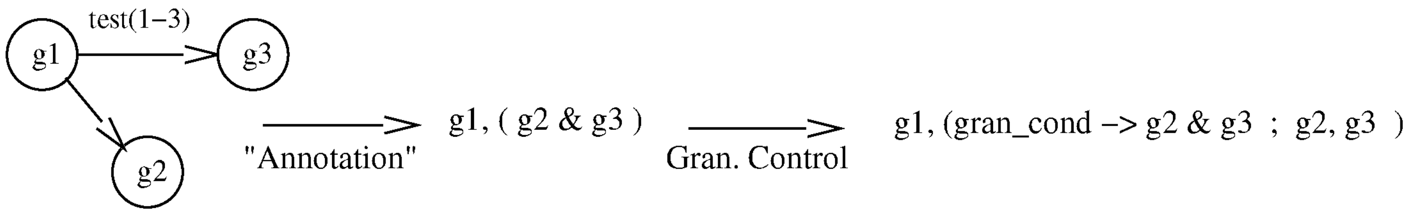
# Refinements (2): Granularity-Aware Annotation

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

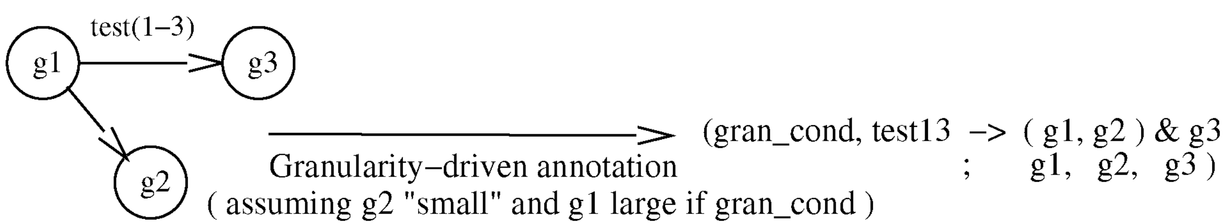


# Refinements (2): Granularity-Aware Annotation

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

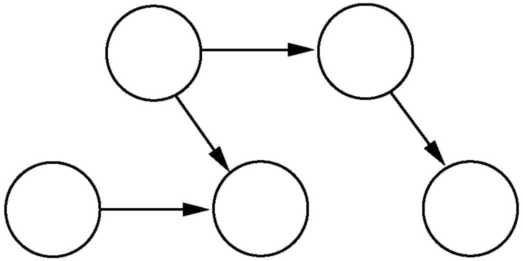



- Developed new annotation algorithm that takes task granularity into account:
  - Annotation is a heuristic process (several alternatives possible).
  - Taking task granularity into account during annotation can help make better choices and speed up annotation process.
  - Tasks with larger cost bounds 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:



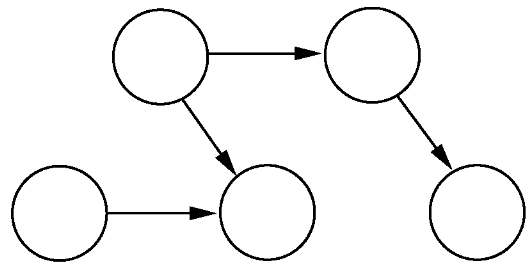
- Assume also that:

$$T(a) < T(c) < T(e) < T(b) < T(d),$$

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:



- 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`.

| | |
|---|---|
| MEL annotator: | `( a, b & c, d & e)` |
| UDG annotator: | `( c & ( a, b, e ), d )` |
| Granularity-aware: | `( a, c, ( b & d ), e )` |

# 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 `nrev` with mode:

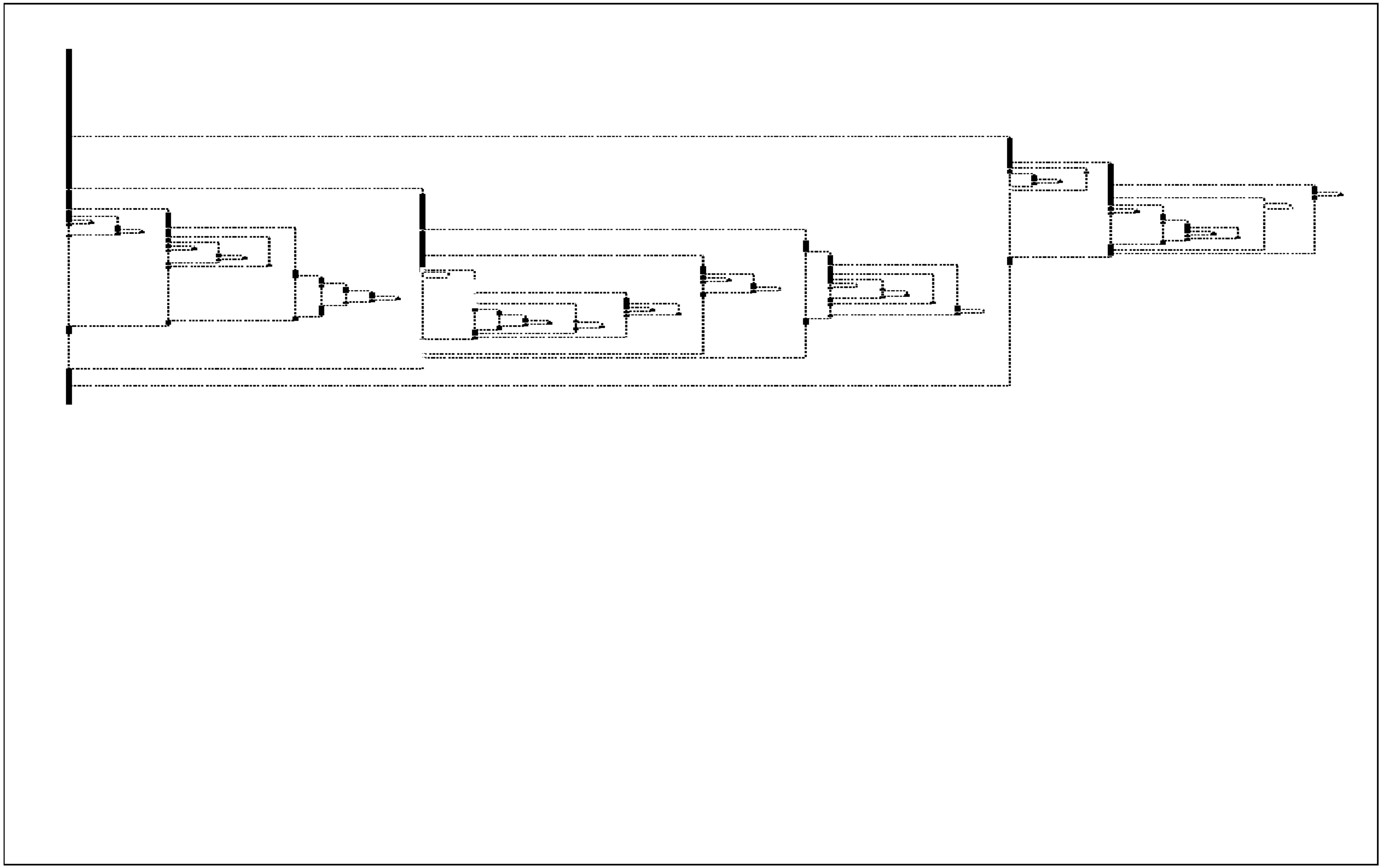
```
:- pred nrev/2 :  list(int) * var.
```

- Estimation of execution time for a concrete input —consider:

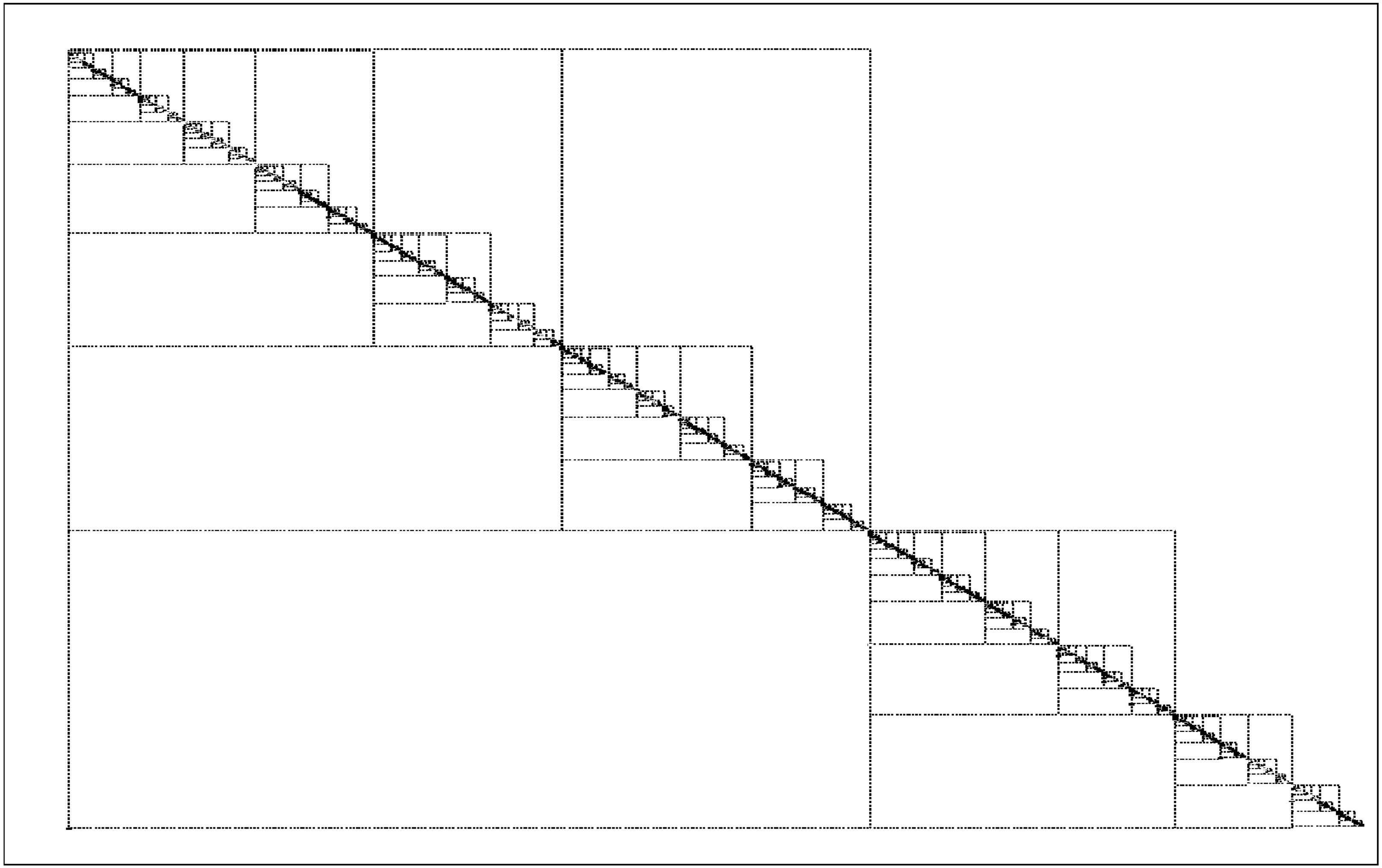
`A = [1,2,3,4,5]`, $\overline{n}$ `= length(A) = 5`

| | Once | Static Analysis | Application | |
|---|---|---|---|---|
| component | $K_{\omega_i}$ | $\mathtt{Cost_p}(I(\omega_i), \overline{n}) = C_i(\overline{n})$ | $C_i(5)$ | $K_{\omega_i} \times C_i(5)$ |
| step | 21.27 | $0.5 \times n^2 + 1.5 \times n + 1$ | 21 | 446.7 |
| nargs | 9.96 | $1.5 \times n^2 + 3.5 \times n + 2$ | 57 | 567.7 |
| giunif | 10.30 | $0.5 \times n^2 + 3.5 \times n + 1$ | 31 | 319.3 |
| gounif | 8.23 | $0.5 \times n^2 + 0.5 \times n + 1$ | 16 | 131.7 |
| viunif | 6.46 | $1.5 \times n^2 + 1.5 \times n + 1$ | 45 | 290.7 |
| vounif | 5.69 | $n^2 + n$ | 30 | 170.7 |
| Execution time $\overline{K_\Omega} \bullet \overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n})$: | | | | 1926.8 |

# Visualization of And-parallelism - (small) qsort, 4 processors

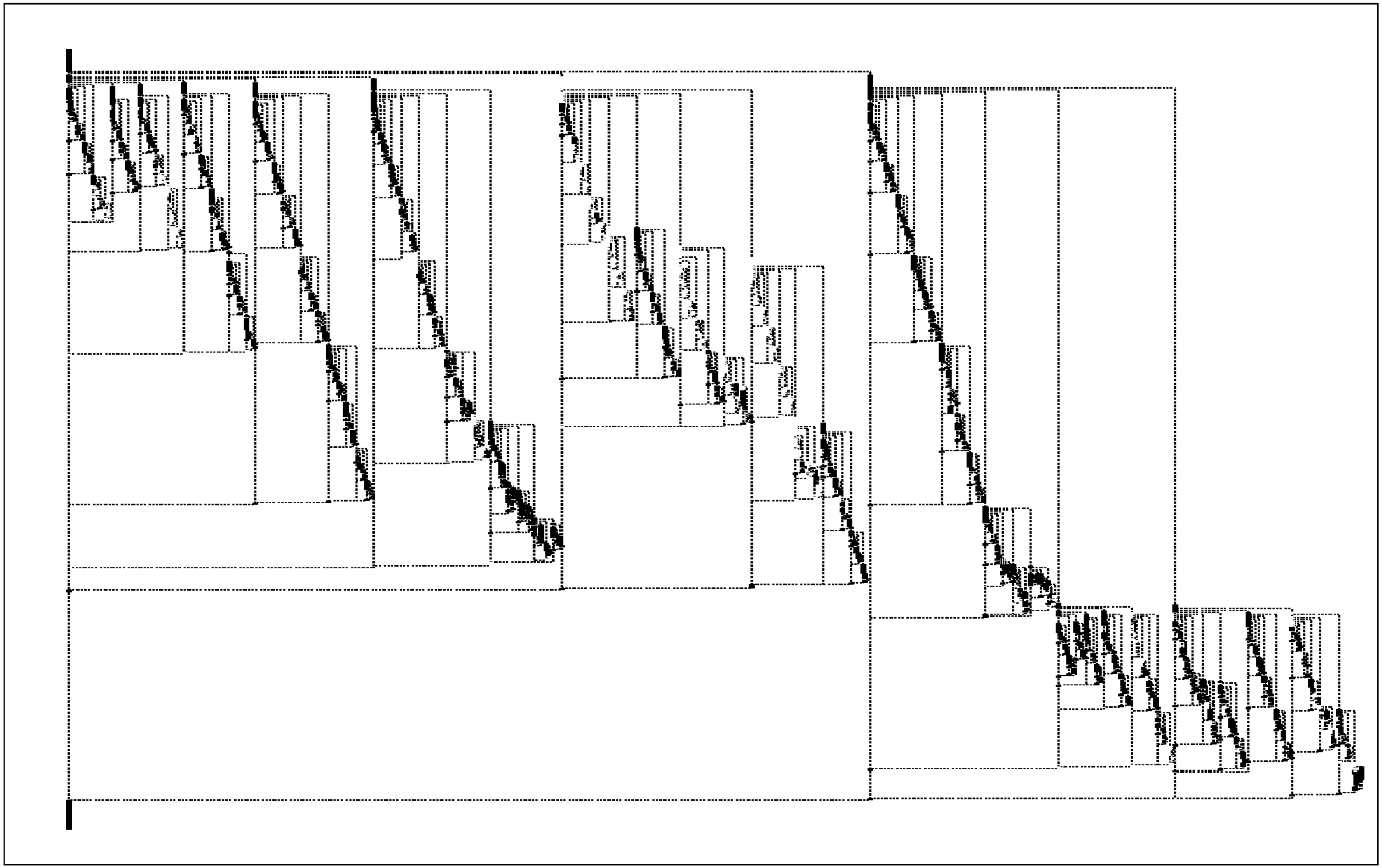


# Fib 15, 1 processor



# Fib 15, 8 processors (same scale)

# Fib 15, 8 processors (full scale)



# Fib 15, 8 processors, with granularity control (same scale)