Towards High-Level Execution Primitives for And-parallelism: Preliminary Results

Amadeo Casas\textsuperscript{1} Manuel Carro\textsuperscript{2} Manuel Hermenegildo\textsuperscript{1,2}

\textsuperscript{1}University of New Mexico (USA)  
\textsuperscript{2}Technical University of Madrid (Spain)
Introduction and motivation

- Parallelism (finally!) becoming mainstream thanks to multicore architectures – even on laptops!
- Declarative languages interesting for parallelization:
  - Program close to problem description.
  - Notion of control provides more flexibility.
  - Amenability to semantics-preserving automatic parallelization.
- Significant previous work in logic and functional programming.
- Two objectives in this work:
  - New, efficient, and more flexible approach for exploiting (unrestricted) (and-)parallelism in LP.
  - Take advantage of new automatic parallelization for LP.
Types of parallelism in LP

- Two main types:
  - Or-parallelism: explores in parallel alternative computation branches.
  - And-parallelism: executes procedure calls in parallel.
    - Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
    - Often marked with &/2 operator: fork-join nested parallelism.
Introduction

Types of parallelism in LP

- **Or-parallelism**: explores in parallel alternative computation branches.
- **And-parallelism**: executes procedure calls in parallel.
  
  - Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
  - Often marked with &/2 operator: fork-join nested parallelism.

**Example (QuickSort: sequential and parallel versions)**

Sequential version:

\[
\text{qsort}([], []). \\
\text{qsort}([X|L], R) :- \\
\text{partition}(L, X, SM, GT), \\
\text{qsort}(GT, SrtGT), \\
\text{qsort}(SM, SrtSM), \\
\text{append}(SrtSM, [X|SrtGT], R).
\]

Parallel version:

\[
\text{qsort}([], []). \\
\text{qsort}([X|L], R) :- \\
\text{partition}(L, X, SM, GT), \\
\text{qsort}(GT, SrtGT), \\
\text{qsort}(SM, SrtSM), \\
\text{append}(SrtSM, [X|SrtGT], R).
\]

- We will focus on and-parallelism.
  
  - Need to detect independent tasks.
Introduction

Background: parallel execution and independence

- **Correctness**: same results as sequential execution.
- **Efficiency**: execution time $\leq$ than seq. program (no slowdown), assuming parallel execution has no overhead.

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

\[
\begin{align*}
\text{main} & : - \quad p(X) :- X = [1,2,3]. \\
& \quad s_1 \quad p(X), \\
& \quad s_2 \quad q(X), \\
& \quad \text{write}(X). \\
& \quad q(X) :- X = [], \text{ large computation}. \\
& \quad q(X) :- X = [1,2,3]. 
\end{align*}
\]

- Fundamental issue: $p$ affects $q$ (prunes its choices).
  - $q$ ahead of $p$ is *speculative*.
- **Independence**: correctness + efficiency.
Related work and proposed solution

- Versions of and-parallelism previously implemented: &-Prolog, &-ACE, AKL, Andorra-I, ...

- They rely on complex low-level machinery:
  - Each agent: new WAM instructions, goal stack, parcall frames, markers, etc.

- Current implementation for shared-memory multiprocessors:
  - Each agent: sequential Prolog machine + goal list + (mostly) Prolog code.

- Approach: rise components to the source language level:
  - **Prolog-level**: goal publishing, goal searching, goal scheduling, “marker” creation (through choice-points), ...
  - **C-level**: low-level threading, locking, stack management, sharing of memory, untrailing, ...

→ Simpler machinery and more flexibility.
Ciao and CiaoPP

- **Ciao**: new generation multi-paradigm language.
  - Supports ISO-Prolog (as a library).
  - Predicates, functions (including laziness), constraints, higher-order, objects, tabling, etc.
  - Parallel, concurrent and distributed execution primitives.

- Preprocessor / environment (CiaoPP):
  - Infers many properties such as types, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, etc.
  - Performs automatic verification of program assertions (and bug detection if assertions are proved false).
  - Performs *automatic parallelization and automatic granularity control*. 
CDG-based automatic parallelization

- **Conditional Dependency Graph**: [TOPLAS’99, JLP’99]
  - Vertices: possible sequential tasks (statements, calls, etc.)
  - Edges: conditions needed for independence (e.g., variable sharing).
- Local or global analysis to remove checks in the edges.
- Annotation converts graph back to (now parallel) source code.

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

```
f1
\[\text{icond}(1-3)\]
f2
\[\text{icond}(1-2)\]
f3
\[\text{icond}(2-3)\]
```

```
( test(1-3) -> ( f1, f2 ) & f3 ; f1, ( f2 & f3 ) )
```

Alternative:
```
Annotation
```

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

```
g1
```

```
g2
```

```
g3
```

```
```

```
```

```
Alternative: f1, ( f2 & f3 )
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
An alternative, more flexible source code annotation

- Classical parallelism operator &/2: nested fork-join.
- However, more flexible constructions can be used to denote parallelism:
  - $G \&> H_G$ — schedules goal $G$ for parallel execution and continues executing the code after $G \&> H_G$.
    - $H_G$ is a *handler* which contains / points to the state of goal $G$.
  - $H_G \prec <$ — waits for the goal associated with $H_G$ to finish.
    - The goal $H_G$ was associated to has produced a solution; bindings for the output variables are available.

- Operator &/2 can be written as:
  $$A \& B : \leftarrow A \&> H, \text{call}(B), H \prec \&.$$

- Optimized deterministic versions: &!/2, &!/1.
Expressing more parallelism

- More parallelism can be exploited with these primitives.
- Take the sequential code below (dep. graph at the right) and three possible parallelizations:

| p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z). | p(X,Y,Z) :- a(X,Z) & c(Y), b(X) & d(Y,Z). | p(X,Y,Z) :- c(Y) & (a(X,Z), b(X)), d(Y,Z). |
| Sequential | Restricted IAP | Unrestricted IAP |

- In this case: unrestricted parallelization at least as good (time-wise) as any restricted one, assuming no overhead.
Shared-Memory Implementation

Low-level support

- **Low-level parallelism primitives:**
  
apll:push_goal(+Goal,+Det,-Handler).
apll:find_goal(-Handler).
apll:goal_available(+Handler).
apll:retrieve_goal(+Handler,-Goal).
apll:goal_finished(+Handler).
apll:set_goal_finished(+Handler).
apll:waiting(+Handler).

- **Synchronization primitives:**
  
apll:suspend.
apll:release(+Handler).
apll:release_some_suspended_thread.
apll:enter_mutex(+Handler).
apll:enter_mutex_self.
apll:release_mutex(+Handler).
apll:release_mutex_self.
Thread creation:

```prolog
create_agents(0) :- !.
create_agents(N) :-
    N > 0,
    conc:start_thread(agent),
    N1 is N - 1,
    create_agents(N1).
```

agent :-
    apll:enter_mutex_self,
    (find_goal_and_execute -> true ;
     apll:exit_mutex_self,
     apll:suspend)
    ,
    agent.

High-level goal publishing:

```prolog
Goal &!> Handler :-
apll:push_goal(Goal,det,Handler),
apll:release_some_suspended_thread.
```
Prolog-level algorithms (II)

- Performing goal joins:

```prolog
Handler <&! :-
    apll:enter_mutex_self,
    (   apll:goal_available(Handler) ->
        apll:retrieve_goal(Handler,Goal),
        apll:exit_mutex_self,
        call(Goal)
    ;
        apll:exit_mutex_self,
        perform_other_work(Handler)
).
```

```prolog
perform_other_work(Handler) :-
    apll:enter_mutex_self,
    (   apll:goal_finished(Handler),
        apll:exit_mutex_self,
        ;
        find_goal_and_execute -> true
    ;
        apll:exit_mutex_self,
        apll:suspend
    ),
    perform_other_work(Handler)
).
```
Search for parallel goals:

```prolog
find_goal_and_execute :-
    apll:find_goal(Handler),
    apll:exit_mutex_self,
    apll:retrieve_goal(Handler,Goal),
    call(Goal),
    apll:enter_mutex(Handler),
    apll:set_goal_finished(Handler),
(    apll:waiting(Handler) ->
        apll:release(Handler)
    ;
    true
),
    apll:exit_mutex(Handler).
```

Casas, Carro, Hermenegildo (UNM, UPM)  Towards High-Level Execution Primitives...  CICLOPS’07 - September 8th
## Preliminary Performance Results

(Preliminary) performance results for restricted and-parallelism (I)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Seq.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAKL</td>
<td>1.00</td>
<td>0.97</td>
<td>1.77</td>
<td>1.66</td>
<td>1.67</td>
<td>1.67</td>
<td>1.67</td>
<td>1.67</td>
<td>1.67</td>
</tr>
<tr>
<td>Ann</td>
<td>1.00</td>
<td>0.98</td>
<td>1.86</td>
<td>2.65</td>
<td>3.37</td>
<td>4.07</td>
<td>4.65</td>
<td>5.22</td>
<td><strong>5.90</strong></td>
</tr>
<tr>
<td>Boyer</td>
<td>1.00</td>
<td>0.32</td>
<td>0.64</td>
<td>0.95</td>
<td>1.21</td>
<td>1.32</td>
<td>1.47</td>
<td>1.57</td>
<td>1.64</td>
</tr>
<tr>
<td>BoyerGC</td>
<td>1.00</td>
<td>0.90</td>
<td>1.74</td>
<td>2.57</td>
<td>3.15</td>
<td>3.85</td>
<td>4.39</td>
<td>4.78</td>
<td><strong>5.20</strong></td>
</tr>
<tr>
<td>Deriv</td>
<td>1.00</td>
<td>0.32</td>
<td>0.61</td>
<td>0.86</td>
<td>1.09</td>
<td>1.15</td>
<td>1.30</td>
<td>1.55</td>
<td>1.75</td>
</tr>
<tr>
<td>DerivGC</td>
<td>1.00</td>
<td>0.91</td>
<td>1.63</td>
<td>2.37</td>
<td>3.05</td>
<td>3.69</td>
<td>4.21</td>
<td>4.79</td>
<td><strong>5.39</strong></td>
</tr>
<tr>
<td>FFT</td>
<td>1.00</td>
<td>0.61</td>
<td>1.08</td>
<td>1.30</td>
<td>1.63</td>
<td>1.65</td>
<td>1.67</td>
<td>1.68</td>
<td>1.70</td>
</tr>
<tr>
<td>FFTGC</td>
<td>1.00</td>
<td>0.98</td>
<td>1.76</td>
<td>2.14</td>
<td>2.71</td>
<td>2.82</td>
<td>2.99</td>
<td>3.08</td>
<td>3.37</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.00</td>
<td>0.30</td>
<td>0.60</td>
<td>0.94</td>
<td>1.25</td>
<td>1.58</td>
<td>1.86</td>
<td>2.22</td>
<td>2.50</td>
</tr>
<tr>
<td>FibonacciGC</td>
<td>1.00</td>
<td>0.99</td>
<td>1.95</td>
<td>2.89</td>
<td>3.84</td>
<td>4.78</td>
<td>5.71</td>
<td>6.63</td>
<td><strong>7.57</strong></td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.00</td>
<td>0.67</td>
<td>1.31</td>
<td>1.82</td>
<td>2.32</td>
<td>2.75</td>
<td>3.20</td>
<td>3.70</td>
<td>4.07</td>
</tr>
<tr>
<td>HanoiDL</td>
<td>1.00</td>
<td>0.47</td>
<td>0.98</td>
<td>1.51</td>
<td>2.19</td>
<td>2.62</td>
<td>3.06</td>
<td>3.54</td>
<td>3.95</td>
</tr>
<tr>
<td>HanoiGC</td>
<td>1.00</td>
<td>0.89</td>
<td>1.72</td>
<td>2.43</td>
<td>3.32</td>
<td>3.77</td>
<td>4.17</td>
<td>4.41</td>
<td><strong>4.67</strong></td>
</tr>
<tr>
<td>MMatrix</td>
<td>1.00</td>
<td>0.91</td>
<td>1.74</td>
<td>2.55</td>
<td>3.32</td>
<td>4.18</td>
<td>4.83</td>
<td>5.55</td>
<td><strong>6.28</strong></td>
</tr>
<tr>
<td>Palindrome</td>
<td>1.00</td>
<td>0.44</td>
<td>0.77</td>
<td>1.09</td>
<td>1.40</td>
<td>1.61</td>
<td>1.82</td>
<td>2.10</td>
<td>2.23</td>
</tr>
<tr>
<td>PalindromeGC</td>
<td>1.00</td>
<td>0.94</td>
<td>1.75</td>
<td>2.37</td>
<td>2.97</td>
<td>3.30</td>
<td>3.62</td>
<td>4.13</td>
<td><strong>4.46</strong></td>
</tr>
<tr>
<td>QuickSort</td>
<td>1.00</td>
<td>0.75</td>
<td>1.42</td>
<td>1.98</td>
<td>2.44</td>
<td>2.84</td>
<td>3.07</td>
<td>3.37</td>
<td>3.55</td>
</tr>
<tr>
<td>QuickSortDL</td>
<td>1.00</td>
<td>0.71</td>
<td>1.36</td>
<td>1.95</td>
<td>2.26</td>
<td>2.76</td>
<td>2.96</td>
<td>3.18</td>
<td>3.32</td>
</tr>
<tr>
<td>QuickSortGC</td>
<td>1.00</td>
<td>0.94</td>
<td>1.78</td>
<td>2.31</td>
<td>2.87</td>
<td>3.19</td>
<td>3.46</td>
<td>3.67</td>
<td><strong>3.75</strong></td>
</tr>
<tr>
<td>Takeuchi</td>
<td>1.00</td>
<td>0.23</td>
<td>0.46</td>
<td>0.68</td>
<td>0.91</td>
<td>1.12</td>
<td>1.32</td>
<td>1.49</td>
<td>1.72</td>
</tr>
<tr>
<td>TakeuchiGC</td>
<td>1.00</td>
<td>0.88</td>
<td>1.61</td>
<td>2.16</td>
<td>2.62</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
</tr>
</tbody>
</table>
(Preliminary) performance results for restricted and-parallelism (II)

(a) Boyer-Moore

(b) Fast-Fourier Transform

(c) Fibonacci

(d) Quicksort

Casas, Carro, Hermenegildo (UNM, UPM) Towards High-Level Execution Primitives... CICLOPS'07 - September 8th
<table>
<thead>
<tr>
<th>Benchm.</th>
<th>And-P</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>FibFunGC</td>
<td>Restricted</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.99</td>
</tr>
<tr>
<td>TakeuchiGC</td>
<td>Restricted</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.88</td>
</tr>
<tr>
<td>FFTGC</td>
<td>Restricted</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.98</td>
</tr>
<tr>
<td>Hamming</td>
<td>Restricted</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.93</td>
</tr>
<tr>
<td>WMS2</td>
<td>Restricted</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Restricted vs. unrestricted and-parallelism (II)

(e) FFT

(f) Hamming

(g) FibFun

(h) Takeuchi

Casas, Carro, Hermenegildo (UNM, UPM) Towards High-Level Execution Primitives... CICLOPS'07 - September 8th
Conclusions and future work

- New implementation approach for exploiting and-parallelism:
  - Simpler machinery.
  - More flexibility.

- Preliminary results:
  - Reasonable speedups are achievable.
  - Additional overhead makes it necessary to perform granularity control.

- Unrestricted and-parallelism:
  - Provides better observed speedups!

- Currently working on:
  - Improving implementation.
  - Developing compile-time (automatic) parallelizers for this approach [LOPSTR’07].