Next Generation of Logic Programming Systems

Gopal Gupta
The University of Texas at Dallas
Acknowledgements

• People:
  • Enrico Pontelli (New Mexico State Univ)
  • Hai-Feng Guo (University of Nebraska) & Karen Villaverde (NMSU)
  • Many others, discussions with whom have influenced our work:
    David H. D. Warren, Vitor Santos Costa, Manuel Hermenegildo, 
    Khayri Ali, Peter Szeredi, Feliks Kluzinak, Mats Carlsson, Ines Dutra, 
    Rong Yang, Tony Beuamont, Ines Dutra, Kish Shen, Raed Sindaha, …
  • Graduate students at UTD: A. Bansal, A. Mallya, L. Simon, Q. Wang

• Funding Agencies:
  • National Science Foundation
  • Department of Energy (Sandia Labs)
  • NATO, Fullbright, JAIST (Japan), State of Texas
Brief History of Parallel LP

- Work on parallel LP began as soon as LP was invented: Pollard (Kowalski’s student) did first thesis in 1981.
- Interest increased with the Japanese 5th Gen. Project:
  - Goal of the FGCS project: to build “fast, intelligent computers”
  - Speed to come from parallel processing
  - Intelligence via AI; realized through LP
- Soon Parallel LP became synonymous with the FGCS project.
The Global FGCS Project

- FGCS spurred global interest in (parallel) LP (MCC, ECRC)
- ECRC produced PEPSys (but, more importantly, produced Constraint Logic Programming).
- MCC produced &-Prolog and spear-headed important work in deductive databases.
- Many other groups got into parallel LP:
  - Bristol, Madrid, SICS, Argonne (Giga Lips project)
- And important systems were produced:
  - Aurora, Muse, &-Prolog, Andorra-I, DDAS, EAM
- Work continued in other groups in the 90s:
  - New Mexico State University
  - U. of Porto/UFRJ
  - Madrid (compile-time analysis)
Global FGCS (cont’d)

• Mistakes that the FGCS made:
  • Commitment to a h/w intensive approach (swept by RISC m/c)
  • Implementors dictated the language:
    • Concurrent Prolog to GHC to Flat GHC to KL1
    • KL1 was too inexpressive & low-level a language for parallelism
    • By late 80s, software impl. of KL1 would beat its hardware impl.

• Lessons to learn:
  • Do not change the language to ease implementation
  • Do not rely on custom hardware (Yes! Use the Intel multicore h/w 😊)
  • The Japanese were ahead of their times; we did not know then how to implement parallel search (or-parallelism) efficiently
    • Therefore, the FGCS project ignored or-parallelism.
  • We now know how to implement or-parallelism efficiently.
Global FGCS Project’s Assumptions

- Exploit parallelism implicitly & from full logic programming.
- Stick to Prolog (Warren): By default, the user should see the same operational semantics as in a sequential implementation.
- No slowdown guarantee (Hermenegildo): High sequential efficiency; parallel overhead should be a fixed factor (< 1).
  - Putting just one more processor should produce a speed-up.
  - We are interested in speed, not speed-ups
  - Implies: do not build your own sequential engine; extend existing ones
- Simplicity of implementation (Gupta): The parallel impl. techniques should be simple; for two reasons:
  - Other people will incorporate them in their system
  - Impl. overhead will be low (easier to guarantee no slowdown)
- No distrib. fat: One feature should not affect another’s perf.
Brief Overview of Our Work

• **Goal**: Exploit parallelism implicitly mainly from symbolic applications by programming them in (C)LP.
• **Symbolic Applications** = Non-numerical applications = Reasoning/NLP/Databases/Compiling/Web/Decision support.
• Applications of LP have been steadily increasing: Learning (ILP), Verification (Tabled LP), Planning (ASP).
• Parallelism from numerical applications can also be exploited (number crunching in Fortran, control in LP)
• **Aim**: to exploit parallelism from 2-20 processors; in the end we also succeeded in building scalable (or-) parallel systems.
Types of Parallelism

- Or-parallelism: multiple matching rules explored in parallel
- IAP: goals that do not share bindings are executed in parallel (equiv. to evaluating args in parallel in FP)
- DAP: goals that share bindings explored in parallel preserving dependencies (equivalent to executing a call and its argument in parallel).

```prolog
qsort([], []).
qsort([P|T], L) :- partition(T, P, A, B),
    qsort(A, L1),
    qsort(B, L2),
    append(L1, [P|L2], L).
```
Types of Parallelism (cont’d)

• Data Or-parallelism: member(X, [1, .., n]) type of calls automatically flattened into a single choicepoint at run time under certain conditions
  • Last Alternative Optimization

• Data And-parallelism: map(P, [1, …, n], R) automatically flattened into a single parcall frame at runtime under certain conditions
  • Last Parallel Call Optimization
Parallel LP Systems

- Large number of systems built:
  - Or-parallelism: Aurora (Bristol), Muse (SICS)
  - IAP: &-Prolog (MCC/Madrid), &-ACE (NMSU)
  - DAP: KL1 (ICOT), Parlog (Imperial), DDAS (Cambridge)
- Challenge: combine all these forms of parallel systems into one
  - Attempted by the ACE system
The ACE System

- Exploits all sources of parallelism
  - Or-parallelism, independent and-parallelism, dependent and-parallelism, data or-parallelism, data and-parallelism + coroutining

- Engine highly optimized (based on SICStus Prolog with many optimizations for parallelism added)

- Massive parallelism was not the aim; desktop multiprocessors (including multicores)

- Shown good performance over a range of programs, many of which are thousands of line long.
The ACE System

- ACE organizes processors in teams (cf: Andorra-I)
  - IAP/DAP exploited within processors in a team
  - Or-parallelism exploited between teams
- Parallel overhead: approximately 5%
- Supports full Prolog
- Ideal for network of distributed shared memory mult.
- Lessons learned from the ACE project:
  - Parallelism can be exploited from symbolic apps
  - And-parallelism harder to exploit in a scalable manner
  - Or-parallelism easier to exploit in a scalable manner
The ACE System: Performance

Figure 1: Speedups in ACE
ACE Performance: Artwork

<table>
<thead>
<tr>
<th>Query</th>
<th>ACE Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Sentence₁</td>
<td>4810</td>
</tr>
</tbody>
</table>

Table 1: Parallel prediction (Sun Sparc, times in ms.)
ACE Performance: Artwork

ArtWork on ACE
(And-parallel Execution)

Figure 2: Speedups for and-Parallel Artwork
ACE Performance: ULTRA

<table>
<thead>
<tr>
<th>Query</th>
<th>ACE Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>English to Chinese</td>
<td>322669</td>
</tr>
</tbody>
</table>

Table 3: Or-parallelism in ULTRA (Sequent, ms.)

<table>
<thead>
<tr>
<th>Goals executed</th>
<th>ACE agents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Eng to Ger</td>
<td>562740</td>
</tr>
<tr>
<td>Eng to Chin</td>
<td>322669</td>
</tr>
<tr>
<td>Eng to Span</td>
<td>91519</td>
</tr>
</tbody>
</table>

Table 4: Execution Times for ULTRA (Sequent Symmetry, times in ms.)
ACE Performance: ULTRA

Ultra on ACE
(And-parallel Execution)

- English to German
- English to Chinese
- English to Spanish

Speedup vs. No. of Processors
Scalable Or-parallelism

- One reason why the Japanese FGCS project failed was the inability to implement or-parallelism efficiently (the first thing to be thrown out).
- Today the multiple environment representation problem is understood well.
- We know how to implement or-parallelism including on scalable parallel machines.
- Stack splitting: generalization of stack-copying in which alternatives are distributed at the time of stack copying.
- Leads to superb performance on all types of parallel m/c.
Stack-splitting Performance

- Parallel overhead: 5-10%; 14 proc. Sun Sparc

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-Costas</td>
<td>715.369</td>
<td>368.298</td>
<td>184.141</td>
<td>92.165</td>
<td>53.453</td>
</tr>
<tr>
<td>Stable</td>
<td>653.705</td>
<td>368.943</td>
<td>185.474</td>
<td>92.811</td>
<td>53.860</td>
</tr>
<tr>
<td>Knight</td>
<td>275.737</td>
<td>141.213</td>
<td>70.528</td>
<td>35.539</td>
<td>22.403</td>
</tr>
<tr>
<td>8-Costas</td>
<td>66.392</td>
<td>34.281</td>
<td>17.192</td>
<td>8.680</td>
<td>5.202</td>
</tr>
<tr>
<td>8-Puzzle</td>
<td>52.945</td>
<td>29.601</td>
<td>15.026</td>
<td>7.845</td>
<td>4.754</td>
</tr>
<tr>
<td>Bart</td>
<td>25.562</td>
<td>15.411</td>
<td>6.868</td>
<td>3.577</td>
<td>2.144</td>
</tr>
<tr>
<td>Solitaire</td>
<td>12.912</td>
<td>7.598</td>
<td>3.813</td>
<td>2.029</td>
<td>1.335</td>
</tr>
<tr>
<td>10-Queens</td>
<td>7.575</td>
<td>3.922</td>
<td>2.087</td>
<td>1.378</td>
<td>1.141</td>
</tr>
<tr>
<td>Hamilton</td>
<td>6.895</td>
<td>3.879</td>
<td>1.940</td>
<td>1.151</td>
<td>0.761</td>
</tr>
<tr>
<td>Map Coloring</td>
<td>2.036</td>
<td>1.298</td>
<td>0.696</td>
<td>0.479</td>
<td>0.430</td>
</tr>
<tr>
<td>8-Queens</td>
<td>0.306</td>
<td>0.198</td>
<td>0.143</td>
<td>0.157</td>
<td>0.149</td>
</tr>
</tbody>
</table>

Table 1: Incremental Stack-splitting (sec.)
Stack-splitting on Beowulf

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costas</td>
<td>412.579</td>
<td>210.228</td>
<td>105.132</td>
<td>52.686</td>
<td>26.547</td>
<td>14.075</td>
</tr>
<tr>
<td>Knight</td>
<td>159.950</td>
<td>81.615</td>
<td>40.929</td>
<td>20.754</td>
<td>10.939</td>
<td>8.248</td>
</tr>
<tr>
<td>Stable</td>
<td>62.638</td>
<td>35.299</td>
<td>17.899</td>
<td>9.117</td>
<td>4.844</td>
<td>3.315</td>
</tr>
<tr>
<td>Send More</td>
<td>61.817</td>
<td>32.953</td>
<td>17.317</td>
<td>8.931</td>
<td>4.923</td>
<td>3.916</td>
</tr>
<tr>
<td>8 Costas</td>
<td>38.681</td>
<td>19.746</td>
<td>9.930</td>
<td>5.052</td>
<td>2.733</td>
<td>1.753</td>
</tr>
<tr>
<td>8 Puzzle</td>
<td>27.810</td>
<td>15.387</td>
<td>8.442</td>
<td>10.522</td>
<td>3.128</td>
<td>5.940</td>
</tr>
<tr>
<td>Bart</td>
<td>13.619</td>
<td>7.958</td>
<td>4.090</td>
<td>2.031</td>
<td>1.600</td>
<td>0.811</td>
</tr>
<tr>
<td>Solitaire</td>
<td>5.909</td>
<td>3.538</td>
<td>1.811</td>
<td>1.003</td>
<td>0.628</td>
<td>0.535</td>
</tr>
<tr>
<td>10 Queens</td>
<td>4.572</td>
<td>2.418</td>
<td>1.380</td>
<td>0.821</td>
<td>1.043</td>
<td>0.905</td>
</tr>
<tr>
<td>Hamilton</td>
<td>3.175</td>
<td>1.807</td>
<td>0.952</td>
<td>0.610</td>
<td>0.458</td>
<td>0.486</td>
</tr>
<tr>
<td>Map Coloring</td>
<td>1.113</td>
<td>0.702</td>
<td>0.430</td>
<td>0.319</td>
<td>0.318</td>
<td>0.348</td>
</tr>
<tr>
<td>8 Queens</td>
<td>0.185</td>
<td>0.162</td>
<td>0.166</td>
<td>0.208</td>
<td>0.169</td>
<td>0.180</td>
</tr>
</tbody>
</table>

Table 4: Timings for Incremental Stack-Splitting (Time in sec.)
Future LP Systems

• LP is a vibrant field: more and more applications are being shown to be elegantly solvable by advanced LP systems:
  • Tabled LP for Verification and Semantic Web apps
  • Inductive LP for Machine Learning apps
  • Constraint LP for Optimization/Search problems
  • Answer Set Programming for Planning and reasoning problems
• These advances have been made independent of each other.
• Challenge for the LP community is to combine these advances into a single system in which parallelism is also exploited.
• Such a system will allow highly complex applications to be developed with unprecedented ease.
Need for Simple Impl. Techniques

- Problem with declarative languages is that their impl. technology is very complex: main reason why multiple advances in LP have not been integrated into one.
- Challenge for implementors: design techniques that are so simple that they can be incorporated in any LP system in a few man months of work.
- Obviously, we have been working on these techniques:
  - Stack splitting for realizing or-parallelism
  - DRA for realizing tabled LP
  - Co-recursion for realizing ASP
  - Continuation trailing for realizing Andorra-I style coroutining
Possible Applications

• We are working on this next generation LP system that combines constraints, tabling, andorra-I, parallelism & ASP

• Significantly complex applications become possible:
  • Model checking of specifications
  • Verification of timed systems (more general type of timed constraints become possible)
  • Complex planning/agent applications including those involving real-time become possible
  • Semantic web applications (e.g., implementations of description logics) can be easily implemented.
  • Bio-informatics applications w/ constraint LP

• In all cases, exploitation of parallelism will result in performance that we think will be significantly better than that of dedicated systems.
Declarative Languages

• As we demonstrate the ease with which declarative languages can solve highly complex problems, declarative languages will eventually prevail.
• Similar to debate between Roman numerals and decimal nos.; it took 100s of years for the world to accept decimal numbers.
• IT industry is gradually moving towards declarative langs:
  • APIs: programming with functions
  • Automatic memory management (in Java, then in C#)
  • (more) logical pointers (i.e., less distinction between pointer & its value; pointer vs reference)
• However, the most critical change needed (single assignment) not adopted yet; may take 50 years 😊
Conclusions

• Parallelism can be exploited implicitly from logic programs.
• Considerable work done in building parallel LP systems.
• Considerable work done in making LP systems suitable for advanced (intelligent) applications (tabled LP, ILP, ASP, constraints).
• The implementation techniques are reasonably well understood and various parallel systems built.
• Considerable progress has been made in building support tools: automatic parallelizers, granularity analyzers, parallel execution visualization tools.
• Future work: develop very simple implementation techniques that will help in combining various advanced LP systems along with parallelism to produce a super powerful, super fast LP system that will

REALIZE THE FGCS DREAM
The field of LP and parallel LP is ready for multicore
5th Gen Project: Reissue the Challenge

1. Advances in LP permit highly advanced (intelligent) apps:
   • Tabled LP for Verification, Semantic Web
   • ASP for planning, non-monotonic reasoning
   • ILP for learning applications
   • Constraint LP for search/optimization applications

2. Inexpensive multi-cores are becoming available, and the LP community knows how to efficiently exploit parallelism

IT'S TIME TO RESTART THE FIFTH GENERATION PROJECT WHICH WILL PUT 1 AND 2 TOGETHER TO OBTAIN INTELLIGENCE AND SPEED
References