Next Generation of Logic Programming Systems

Gopal Gupta

The University of Texas at Dallas

ALPS LAB @ UTD

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

ALPS LAB @ UTD

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.

SEPTEMBER 1983

VOLUME 26 NUMBER 9

COMMUNICATIONS OF THE acm

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).

```
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

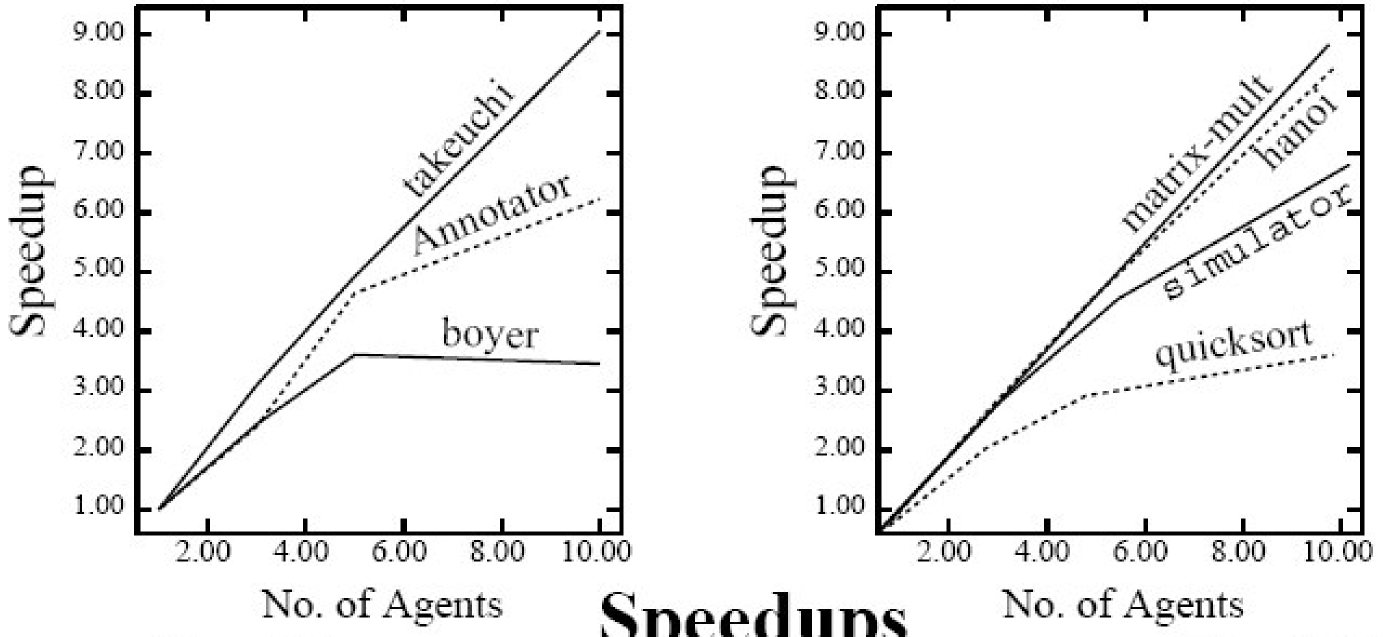
The ACE System

- 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



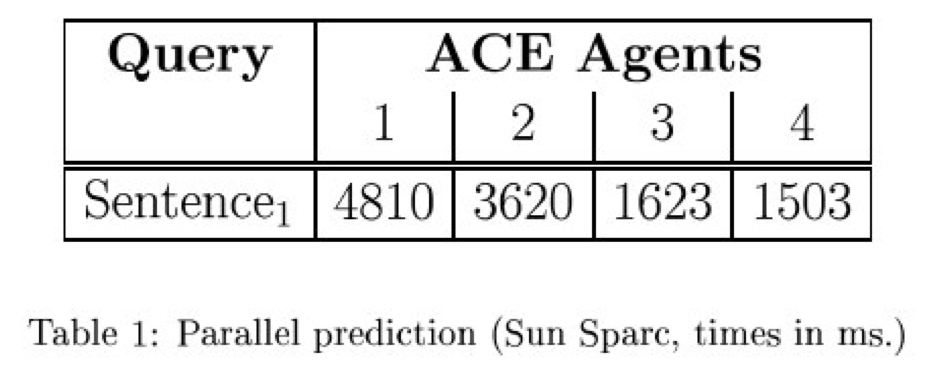
Speedups

Fig (i)

Fig (ii)

Figure 1: Speedups in ACE

ACE Performance: Artwork



Query	**ACE Agents**			
	1	2	3	4
Sentence$_1$	4810	3620	1623	1503

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

ACE Performance: Artwork

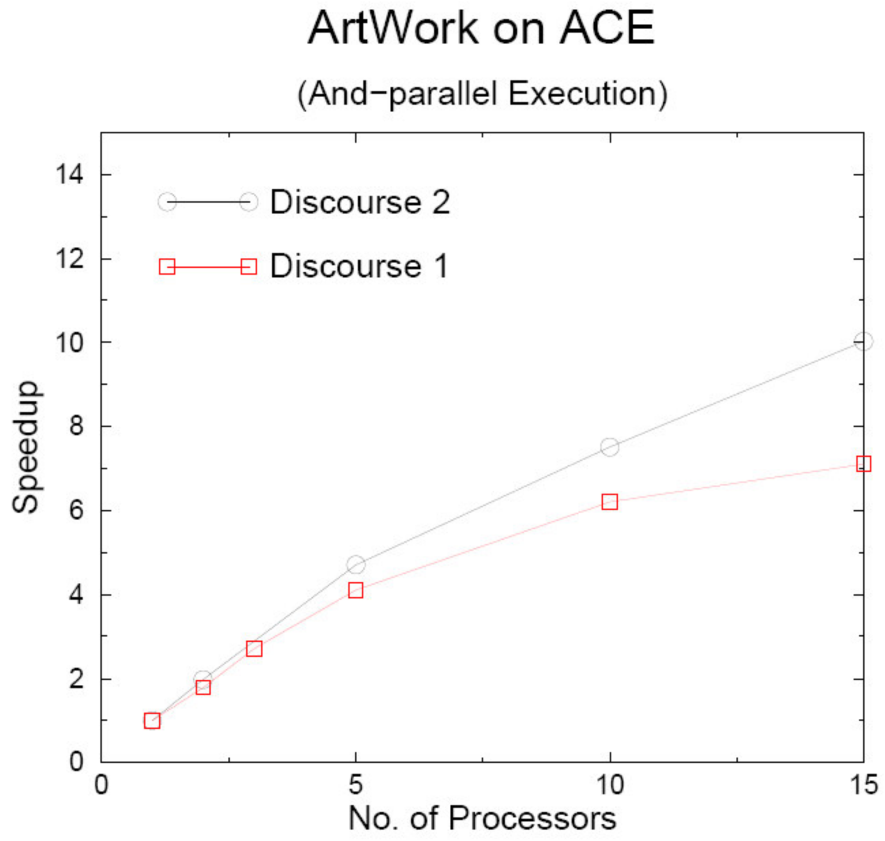
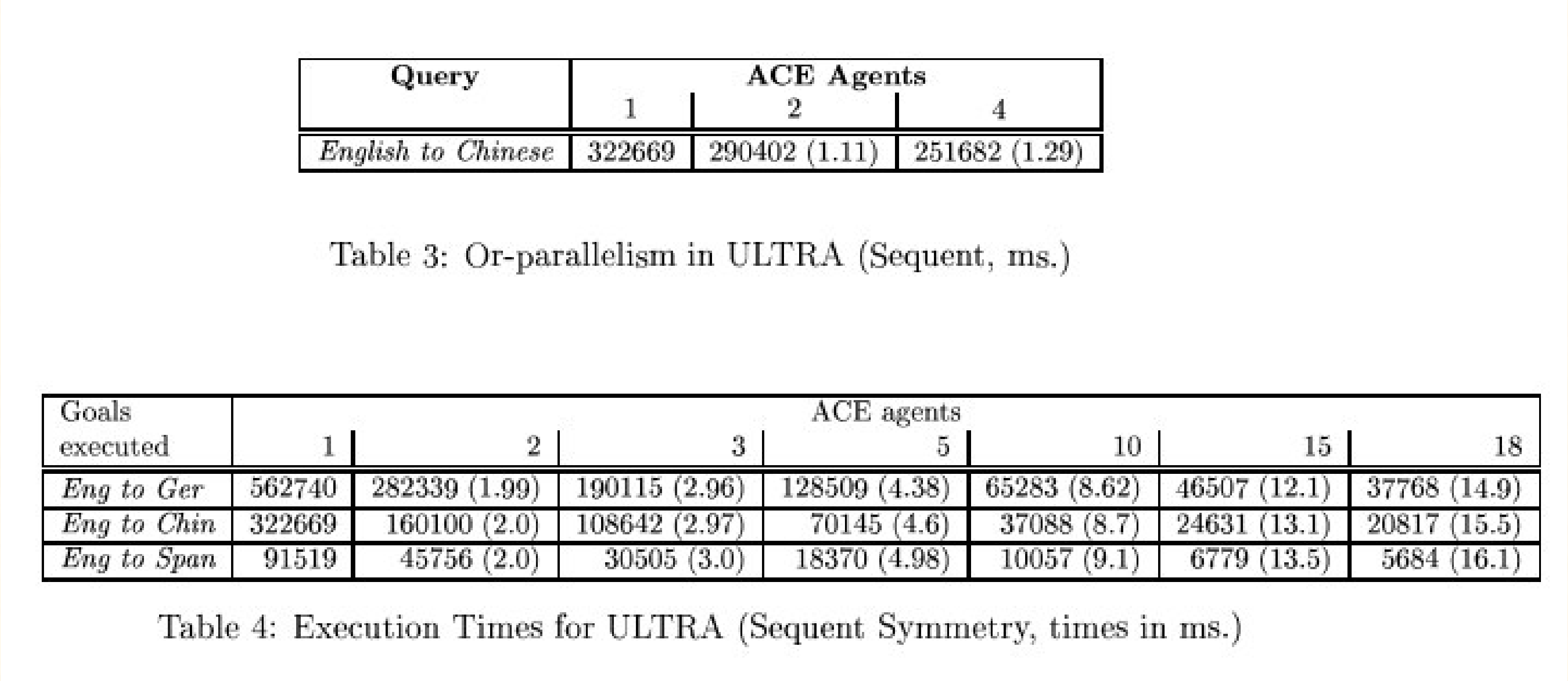


Figure 2: Speedups for and-Parallel Artwork

ALPS LAB @ UTD

ACE Performance: ULTRA



Query	**ACE Agents**		
	1	2	4
English to Chinese	322669	290402 (1.11)	251682 (1.29)

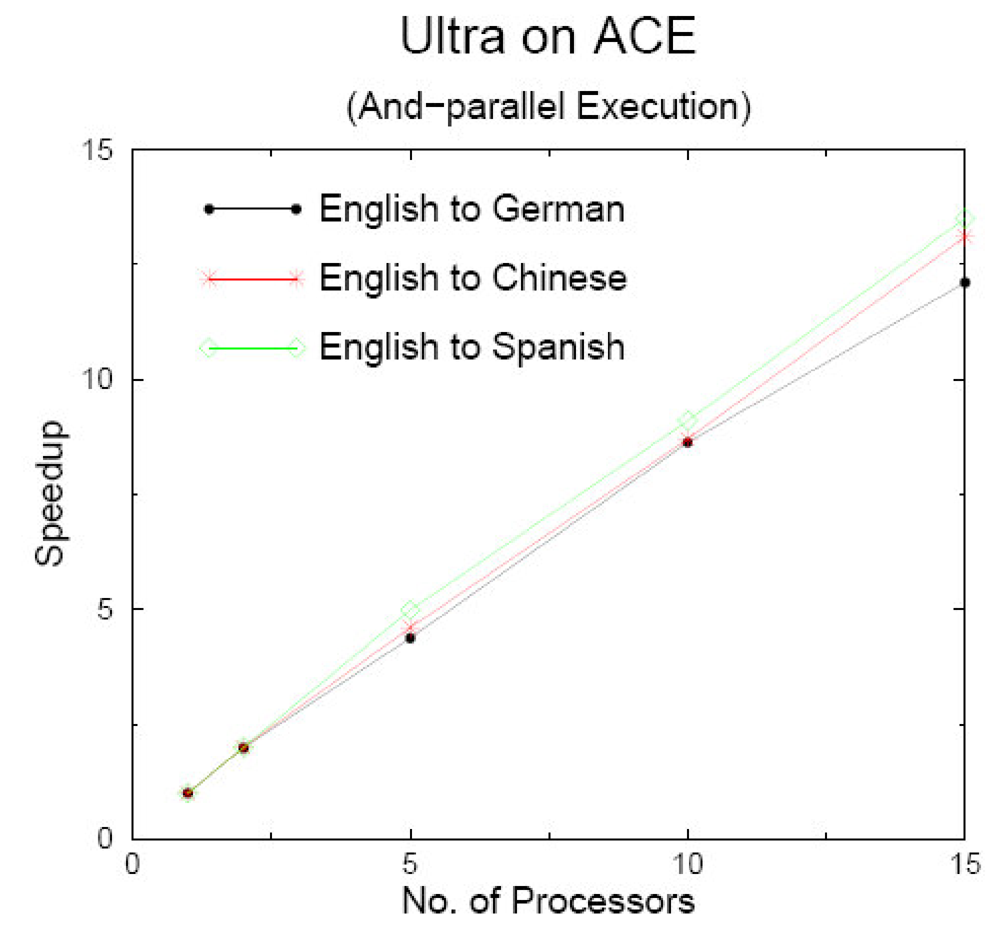
Table 3: Or-parallelism in ULTRA (Sequent, ms.)

Goals executed	ACE agents						
	1	2	3	5	10	15	18
Eng to Ger	562740	282339 (1.99)	190115 (2.96)	128509 (4.38)	65283 (8.62)	46507 (12.1)	37768 (14.9)
Eng to Chin	322669	160100 (2.0)	108642 (2.97)	70145 (4.6)	37088 (8.7)	24631 (13.1)	20817 (15.5)
Eng to Span	91519	45756 (2.0)	30505 (3.0)	18370 (4.98)	10057 (9.1)	6779 (13.5)	5684 (16.1)

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

ALPS LAB @ UTD

ACE Performance: ULTRA



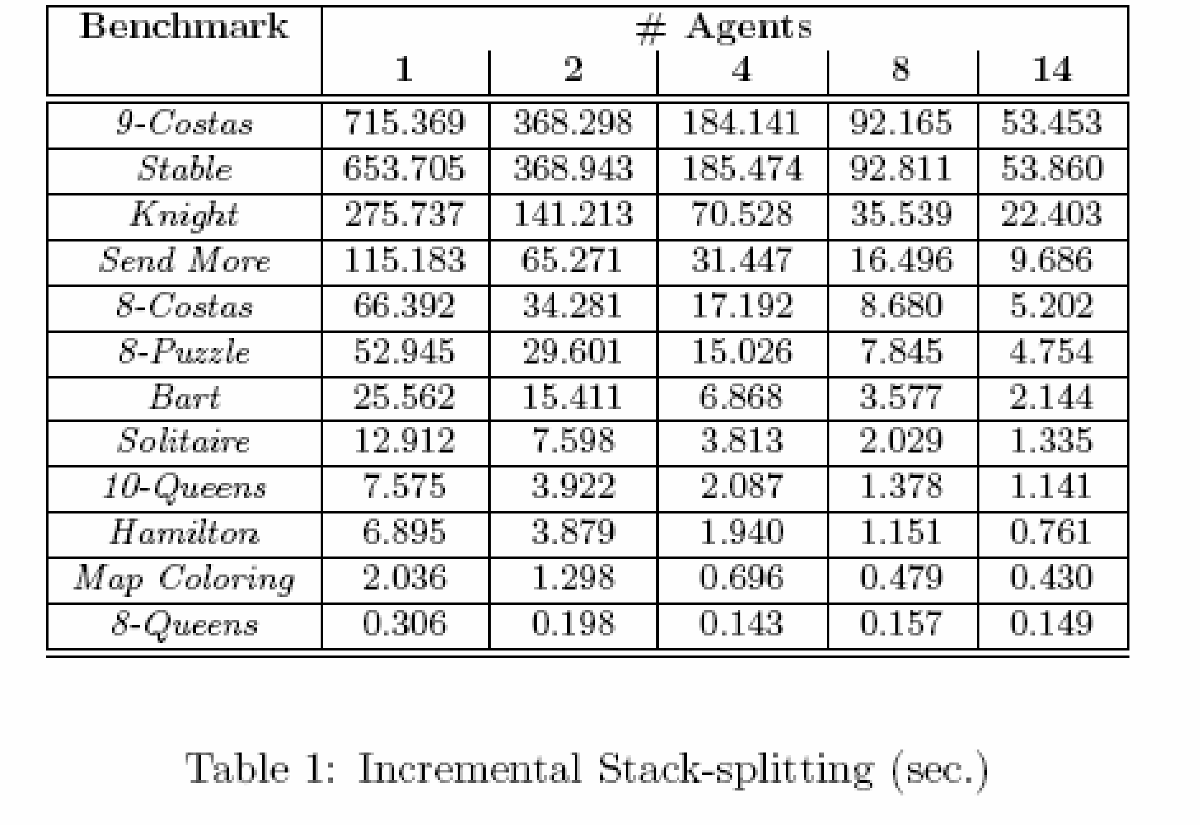
ALPS LAB @ UTD

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

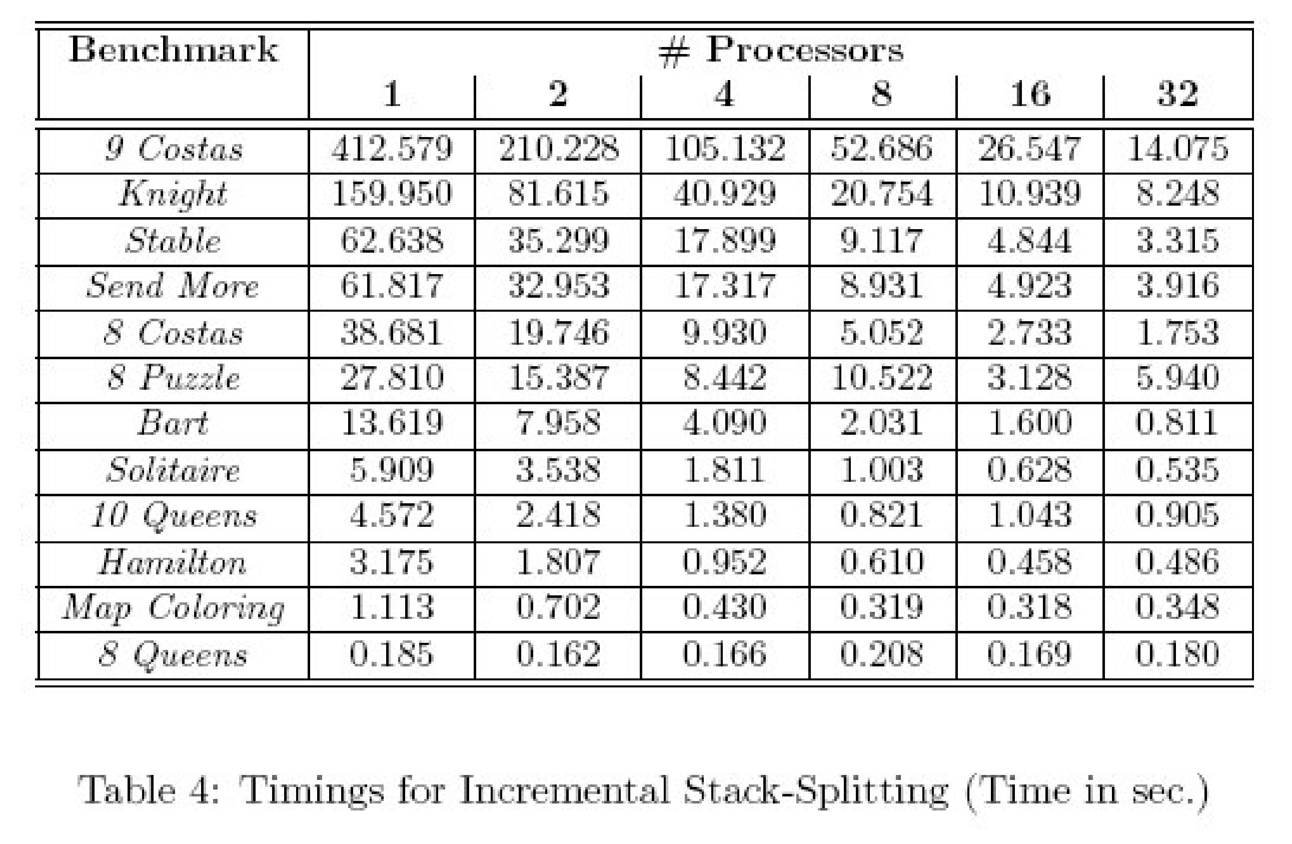


Benchmark	**# Agents**				
	1	**2**	**4**	**8**	**14**
9-Costas	715.369	368.298	184.141	92.165	53.453
Stable	653.705	368.943	185.474	92.811	53.860
Knight	275.737	141.213	70.528	35.539	22.403
Send More	115.183	65.271	31.447	16.496	9.686
8-Costas	66.392	34.281	17.192	8.680	5.202
8-Puzzle	52.945	29.601	15.026	7.845	4.754
Bart	25.562	15.411	6.868	3.577	2.144
Solitaire	12.912	7.598	3.813	2.029	1.335
10-Queens	7.575	3.922	2.087	1.378	1.141
Hamilton	6.895	3.879	1.940	1.151	0.761
Map Coloring	2.036	1.298	0.696	0.479	0.430
8-Queens	0.306	0.198	0.143	0.157	0.149

Table 1: Incremental Stack-splitting (sec.)

ALPS LAB @ UTD

Stack-splitting on Beowulf



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

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

Message

The field of LP and parallel LP is ready for multicores

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

ITS TIME TO RESTART THE FIFTH GENERATON
PROJECT WHICH WILL PUT 1 and 2 TOGETHER
TO OBTAIN **INTELLIGENCE AND SPEED**

SEPTEMBER 1983

VOLUME 26 NUMBER 9

COMMUNICATIONS OF THE acm

References

- G. Gupta, E. Pontelli, K. Ali, M. Carlsson, M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. ACM Transactions on Programming Languages and Systems, Vol 23, No. 4, pp. 472-602.
- G. Gupta. Next Generation Logic Programming Systems. Technical Report, UT Dallas. 2003.
- E. Pontelli. High Performance Logic Programmng. Ph. D. Thesis, NMSU, 1997.
- H-F Guo. High Performance Parallel Prolog Systems. Ph.D. Thesis, NMSU, 2000.
- K. Villaverde. Scalable Parallel Prolog on the Beowulf. Ph.D. thesis. NMSU, 2003.