Parallel Prolog Experiments

Tim Lindholm

Quintus Computer Systems, Inc
Mountain View
California

currently at Argonne National Laboratory

Prolog Characteristics

- Declarative
  - "What" not "How"
  - Predicate logic
- Small set of key features
  - Relational
  - Facts and rules
  - Pattern matching
  - Recursive data structures
  - Internal database
- Concise and compact

Prolog Productivity in Application Design

- "AI" techniques -- functionality
- Rapid prototyping
- High level application specification
- Incremental refinement

Prolog Productivity in Application Development

- Easier to write and debug applications
- Allows concentration on problem
- Uniform approach to information manipulation
- Interactive development environment
- Libraries, toolkits and interfaces, training

Prolog Productivity in Application Deployment

- Time to market
- High performance & efficient memory utilization
- General purpose hardware platforms
- Integratable with other tools
- Effective runtime environments
- Robust and well supported products

Prolog Productivity in Application Maintenance

- Understandable
- Compact
- Modular
- Extensible
- Verifiable

Prolog Users

- Universities
- Research institutions
- Government agencies
- Corporate
  - AI groups
  - Research and development
  - MIS
- System integrators / application developers

Prolog Application Markets

- Manufacturing (aerospace, automobile, electronics)
- CAD (electronic, mechanical, architectural)
- Database, decision support
- CASE

Prolog Application Areas

- Knowledge based systems
  - Fault analysis
  - Diagnosis
  - Configuration
  - Monitoring complex situations
- Components of traditional applications
  - Design
  - Intelligent front ends
  - Compilers, generators
  - Translators

Industry Trends

- Utilization of PCs and technical workstations
- Rapid price and performance improvements
- Distributed networks, distributed computing
- Standardization
  - Languages
  - Operating systems
  - User interfaces
  - Communications
  - Databases
- General purpose hardware
- Multiprocessing and parallelism

Prolog Overview

The Basic Programming Structures are Facts and Rules

```
flight( 'New York',   'San Francisco' ).
flight( 'Washington', 'Chicago' ).
flight( 'Washington', 'Dallas' ).
flight( 'Dallas',     'San Francisco' ).

| ?- flight( Originate, 'San Francisco' ).

Originate = 'New York' ;

Originate = 'Dallas'
```

Prolog Overview

The Basic Programming Structures are Facts and Rules

```
travel(A,B) :-
        flight(A,B).
travel(A,B) :-
        flight(A,Intermediate),
        travel(Intermediate,B).

| ?- travel( Originate, 'San Francisco' ).

Originate = 'New York' ;

Originate = 'Dallas' ;

Originate = 'Washington'
```

Parallelism in Prolog Programs

Why worry about parallelism?

- Expressiveness
  - coroutines?
- Functionality
  - transaction servers?
- Speed

Parallelism in Prolog Programs

Sources of Parallelism in Prolog Programs

- OR-parallelism - investigate multiple alternatives in parallel
- AND-parallelism - solve multiple goals in parallel

...and a swarm of others...

For example:

```
travel(A,B) :-
        flight(A,B).
travel(A,B) :-
        flight(A,Intermediate),
        travel(Intermediate,B).
```

The basic problem: resolving binding conflicts for shared variables

Parallelism in Prolog Programs

Shared Variables

Due to OR-parallelism:

```
travel(A,B) :-
        flight(A,B).
travel(A,B) :-
        flight(A,Intermediate),
        travel(Intermediate,B).
```

Due to AND-parallelism:

```
travel(A,B) :-
        flight(A,B).
travel(A,B) :-
        flight(A,Intermediate),
        travel(Intermediate,B).
```

Parallelism in Prolog Programs

Exploitation of AND-parallelism

Unrestricted AND-parallelism

- Explicit parallelism
- Plenty of parallelism in most applications
- New languages (Parlog, GHC, Concurrent Prolog)
- New implementation techniques needed
- "Porting" existing Prolog applications means rewriting
- New applications cannot take advantage of Prolog installed base

Restricted AND-parallelism

- Exploitation of implicit parallelism?

Parallelism in Prolog Programs

Exploitation of OR-parallelism

Implicit OR-parallelism in Prolog programs

- Exploitation of implicit parallelism
- Plenty of parallelism in a wide class of applications
- Retain Prolog syntax and semantics
- Prolog implementation technology carries over
- Minimal or no changes needed to run existing applications
- Easy porting of new applications across a wide variety of platforms

Caveat: Some algorithmic changes may be needed to take best advantage of parallel execution

Parallelism in Prolog Programs

Claim: OR-parallelism should be attractive to the Prolog vendor and the application developer working in Prolog. To the Prolog implementor, it should be viewed as an implementation detail, like an optimizing compiler.

The Gigalips Project

Participants:

- Manchester University
- Argonne National Laboratory (ANL)
- Swedish Institute of Computer Science (SICS)

Goals:

- Investigate implicit parallelism in Prolog programs
- Target general-purpose shared memory multiprocessors
- Run real programs

The ultimate goal of the Gigalips Project is to run Prolog programs faster than the best sequential systems on shared memory multiprocessors

Aurora

Aurora - a prototype Prolog system exploiting OR-parallelism

"Workers" explore the Prolog search tree in OR-parallel

- the "engine"
- the "scheduler"

The Aurora implementation environment:

- Engine-scheduler interface
- Scheduler test harness
- Instrumentation

Aurora

Aurora's Engine

- Based on SICStus Prolog 0.3
  - Moderately high performance
  - Portable (written in C)

- Runs David H. D. Warren's "SRI model"
  - Creation, accessing variable bindings remain constant time
  - Process creation is inexpensive
  - Task switching can be expensive

Aurora

Aurora's Schedulers

Early schedulers (ANL) relied on global "dispatching pools"

Current schedulers operate on the basis of local information

The various Aurora schedulers:

- ANL scheduler
- Manchester scheduler
- "Wavefront" scheduling (under development at SICS)

Task switching under the SRI model makes scheduling technology critical

Language details also depend on scheduling technology

Aurora

Current Status

Can run moderate-sized "dusty-deck" Prolog programs

Can demonstrate speedups as workers are added

Needs more efficient, robust engine, better memory management

Needs work on scheduling, primitives

Conclusions from Aurora

Engine overhead due to SRI model and scheduler hooks:
15-35%

This overhead defines breakeven with sequential systems

Speedups Measured under Aurora:

Example	speedup for N workers 3	5	
parse5	(2.83)	(4.08)	!!
8-queens2	(2.97)	(4.88)	!!
salt&must	(2.87)	(4.82)	!!
parse3*20	(2.09)	(2.30)	??
farmer*100	(1.63)	(1.69)	??

Speedups measured on a six processor Sequent Balance

Implications for Commercial Prolog Systems

- Quintus Prolog has been released for the Sequent Symmetry
- Studies at ANL indicate that degradation due to the SRI model for a worker based on a higher-performance Quintus Prolog engine would be comparable to those seen in Aurora (and probably not better)
- Together with this, the speedups demonstrated by Aurora allow us to predict performance of a Quintus-based OR-parallel system on the Symmetry
- Critical scheduler technology must continue to develop to make speedups widely accessible, but adherence to the standard interface allows tracking of that technology

The Bottom Line: For a wide class of applications, an OR-parallel Prolog system for the Sequent Symmetry based on Quintus Prolog can be cost-effective.