SAT Solvers for Investigation of Architectures for Cognitive Information Processing

Richard A. Lethin, James R. Ezick,
Samuel B. Luckenbill, Donald D. Nguyen,
John A. Starks, Peter Szilagyi
Reservoir Labs, Inc.

Sponsored by DARPA in the ACIP Program,
William Harrod, IPTO, Program Manager

Thomas Bramhall, AMCOM, Agent

Overview

- Demand for Cognitive Processing
- Historical Architectures for AI / Cognitive Processing
- SAT Solvers as a Cognitive Application
- Application Specific Hardware
- Parallelizing SAT
- Current Performance
- Architectural Implications

Demand for Architectures for Cognitive Processing

- **Signal/Knowledge Processing Algorithms**
- **Workload in the "Knowledge Processing Section" comparable to the "Signal Processing Section"**
- **Problem: how can we make "Knowledge Processing" more efficient through better architectures?**
- **Study SAT Solvers as an example.**

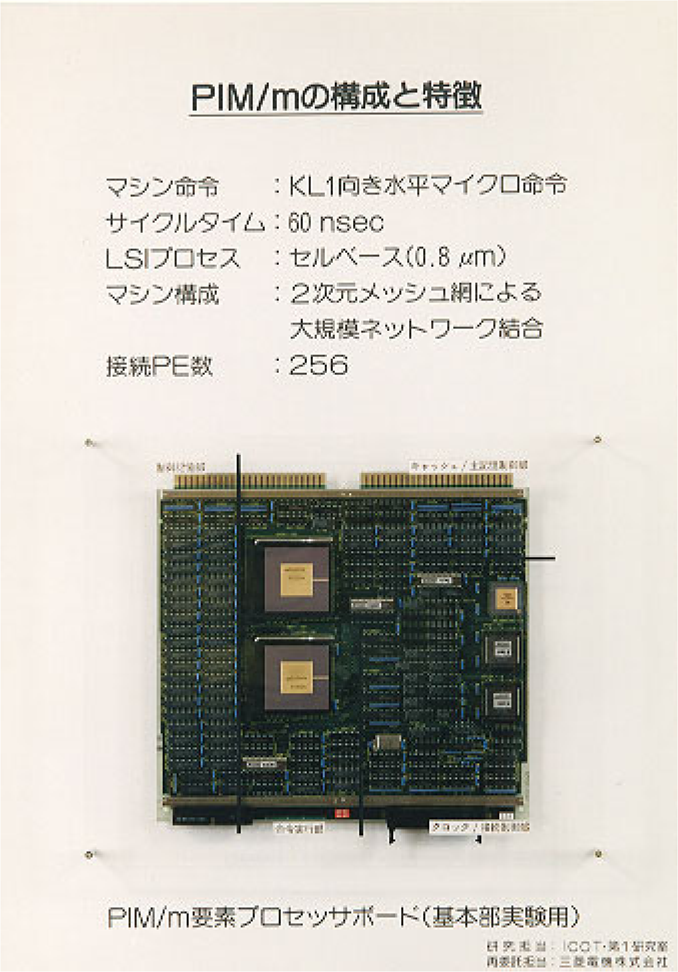
Computer Architectures for AI

Symbolics, 1986

Lisp Machines, ~1983

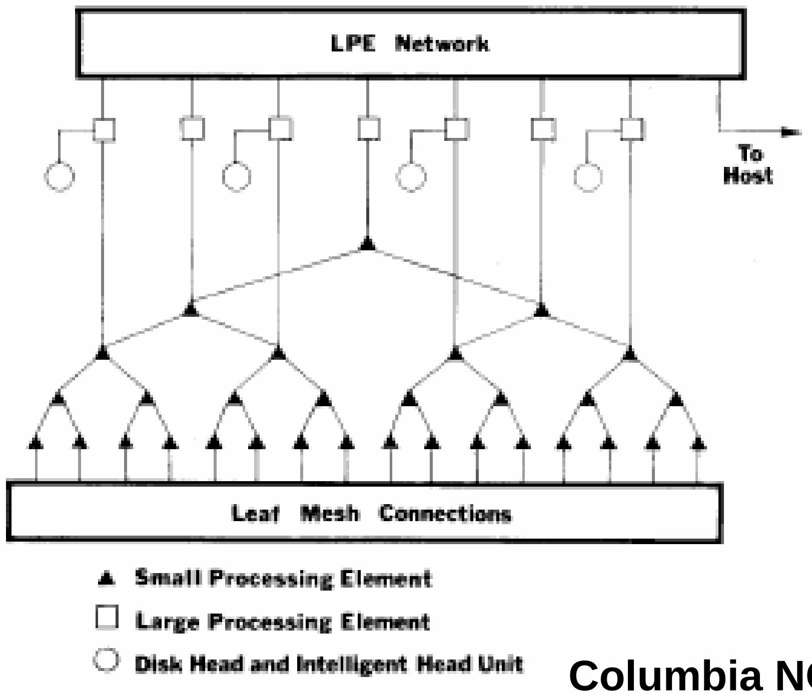
Source: http://en.wikipedia.org/wiki/Lisp_machines

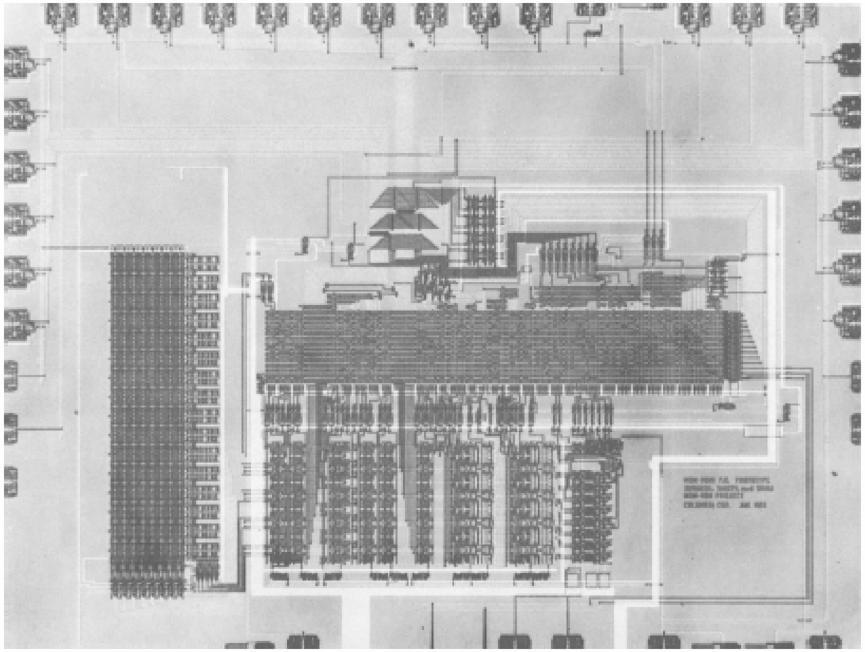
Computer Architectures for AI



Fifth Generation Computing Systems Project, ICOT, 1982-1993

Computer Architectures for AI





Columbia NON-VON 1981-1987

Independent SIMD engines

Massive Parallelism

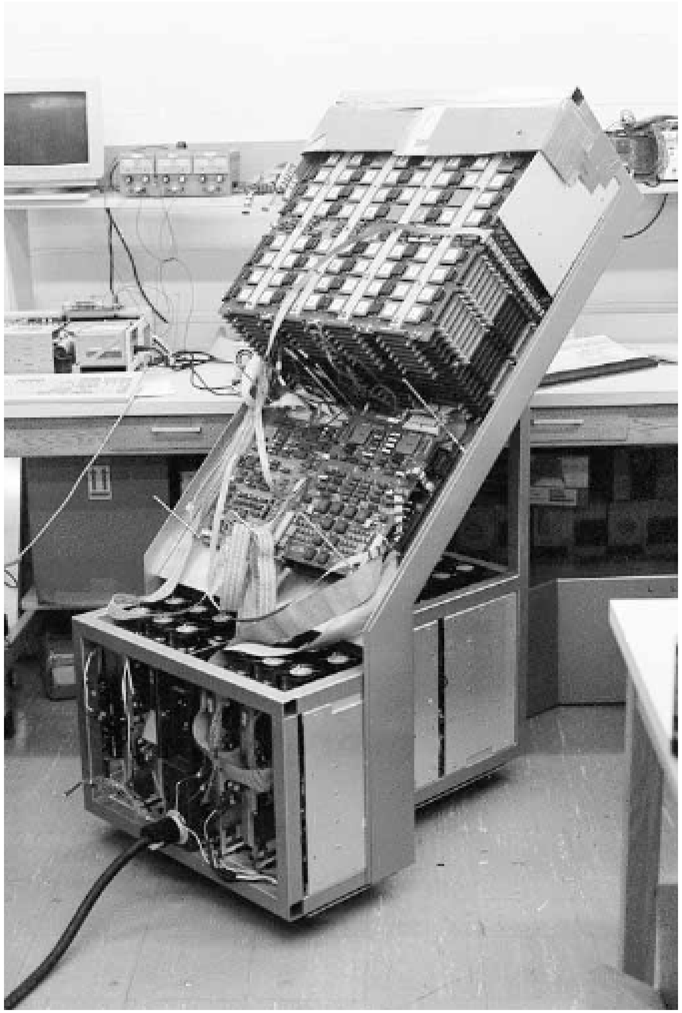
VLSI and network efficiency

Multiple AI algorithms

Computer Architectures for AI

Connection Machine 2 (1988)
Massive Parallelism
Massive Interconnect
SIMD

Computer Architectures for AI



J-Machine, MIT 1993
MIMD
Message-Driven Processors
VLSI and network efficiency

Common Architecture Themes

- Parallelism
- Fine Grained Architectures
- High Performance Networking (Bisection, Latency)
- MIMD or SIMD
- Distributed Memory
- Specialized Operators

- What's changed?

The Satisfiability Problem (SAT)

Definition: Given a Boolean formula E, decide if there is some assignment to the variables in E such that E evaluates to *true*

Example: $E = (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee x_3)$

Solution: E evaluates to *true* (is satisfied) if $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$

- **Recent significant progress → 1,000,000 variables**
- **Some SAT Applications**
  - **Planning**: Route planning; mission planning
  - **Software Verification**: Verifying numerical precision and interprocedural control flow; assertion checking; proving that an implementation meets a specification
  - **Hardware Verification**: Bounded model checking; test pattern generation

SAT as a Cognitive Application

- **Naive algorithm (Davis-Putnam)**
  - Complete backtracking exploration of the search space
- **Improved algorithm (Davis-Putnam-Logemann-Loveland)**
  - Introduced Boolean Constraint Propagation (BCP)
    - Based on unit clause rule
  - Still explores multiple paths through unsatisfiable space
- **Modern SAT algorithms (up to millions of clauses recently)**
  - Conflict resolution
    - Uses learning to prune unsatisfiable search space
  - Non-chronological backtracking
    - Integrated with conflict resolution
    - Prevent solver from exploring unsatisfiable search space
  - Decision heuristics
    - Choose most active variables to assign
  - Restarts
    - Can allow solvers to escape local minima
- **Search is a fundamental AI problem**

SAT as a good application for AI architecture research

- **Distills what we think of as an "AI application" into an apparently simple problem.**
  - Search, mixed with following inferences
  - Big databases of clauses
  - Irregular access to a big database
  - Learning!
  - Data-dependent control flow
  - Arbitrary amounts of parallelism
  - Independent threads going in seemingly different directions

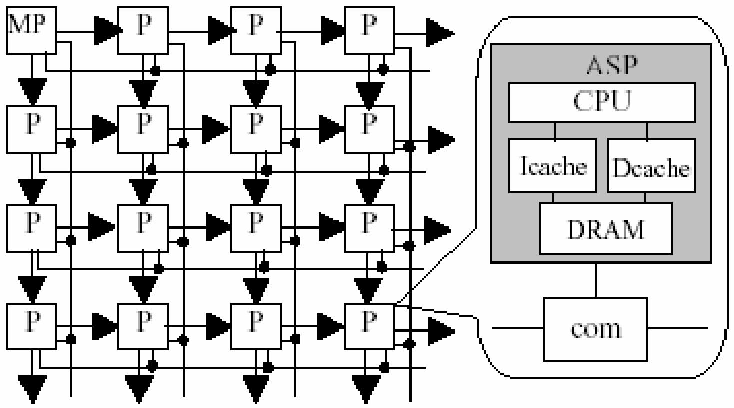
- **But… the simplicity is deceptive!**

General Architecture Techniques for Accelerating Applications

- **Functional Concurrency** - Distinct computational tasks are executed in parallel on different pieces of hardware.
- **Data Concurrency** - Data is partitioned and operated on in parallel by multiple processors.
- **Thread Concurrency** - Sequential sequences of instructions are partitioned and executed in parallel.
- **Special Representations** - Special data formats hold application-specific data efficiently for increased performance.
- **Special Operators** - Special operators accelerate operations which frequently occur in an application.
- **Pipelining** - Data or instructions are moved in lock step through stages of a sequential operation.
- **Data choreography** - Hardware is laid out so as to minimize the physical movement of the data between computational units.
- **Circuit Techniques** - Circuit techniques exist to minimize latency or power consumption, or to maximize throughput or clock frequency.

Prior Work on Specialized Architectures for SAT

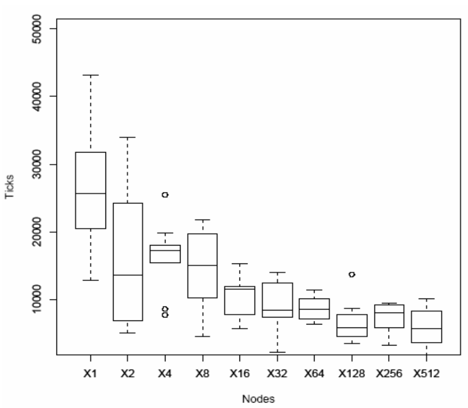
- **FPGAs**
  - Requires chip area and compile time proportional to size of problem.
  - Poor choice: does not scale well.
- **Princeton Architecture (Zhao/Malik)**
  - Tensilica cores customized as processing and routers with special operators.
  - On-chip network in a torus network.
  - Embedded DRAM to store distributed data
  - Similar to MIT's RAW PCA architecture.
  - Takes advantage of fine-grained parallelism to 60x performance improvement.
- **LAN-Based Parallel Solvers**
  - Take advantage of coarse-grained parallelism for 1x to 20x performance improvement.
  - Replication of data limits ability to solve very large problems.



Special-purpose SAT architecture
Source: Zhao01

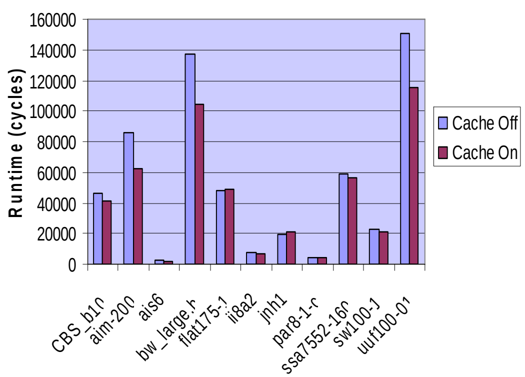
Simulated Results

Coarse-grained parallelism accelerates SAT



- If the network is low-latency or the clause database is replicated per node, additional parallel threads reduce runtime.
- Simulations show between 1x and 20x performance improvement from coarse-grained parallelism (similar to GridSAT and PaSAT).

Caching allows for scaling of parallel SAT



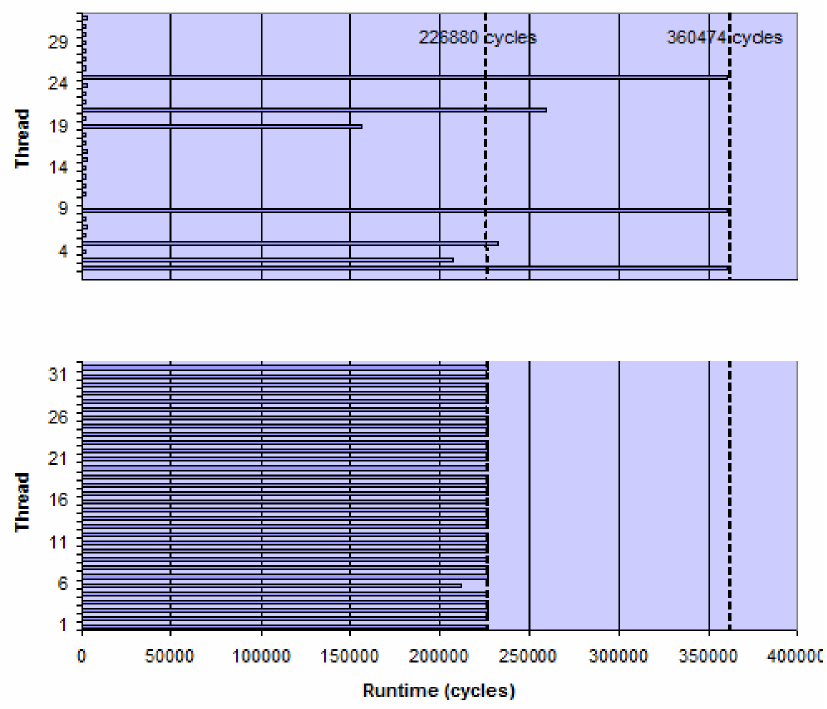
- For higher-latency networks with a distributed clause database, caching often helps to reduce runtime.
- Occasionally, caching increases runtime because the search path changes.

Simulated Results (continued)

Load balancing is critical to distributed parallel SAT

Work Stealing Off

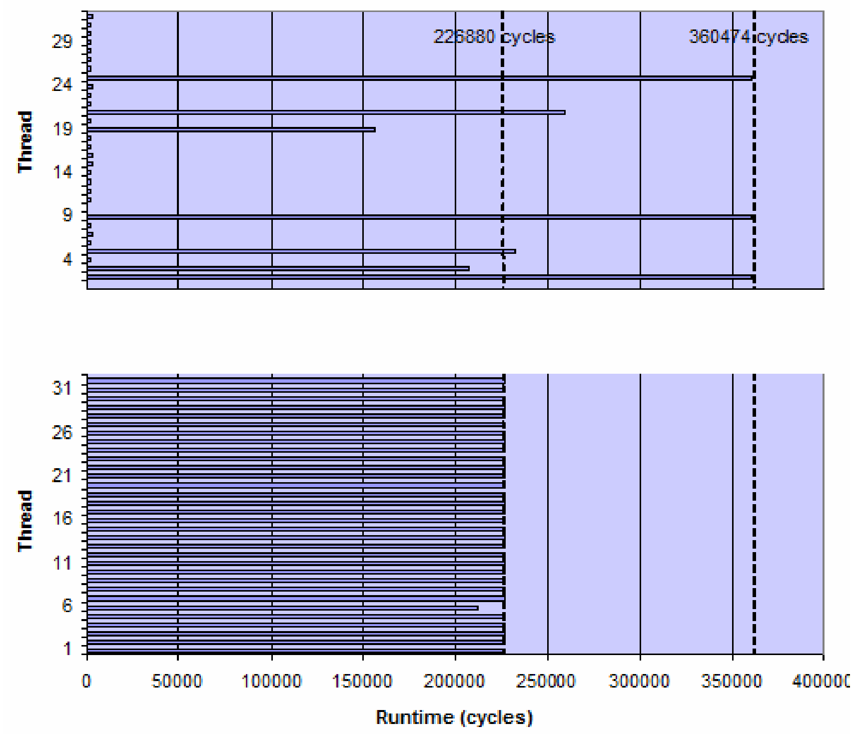
Work Stealing On



- Without load balancing, most threads die immediately and remain idle for the remainder of the search.
- A work-stealing strategy keeps threads active and significantly reduces search time (in this example by 37%).

Load Balancing

- **Runtime of search threads is highly variable**
  - Load balancing is critical to keeping threads active
  - Reduces search time significantly
  - Uses random victim strategy
- **Heuristic: try to share a substantial amount of productive work**
  - Branch low in the assignment stack
  - Branch on variables where the decision heuristic was "unsure"
- **Implemented with minimal overhead for victim thread**
  - Merge low decision levels to prevent unnecessary backtracking



Runtime of search threads with and without load balancing. With load balancing, total search time reduced by 37% for this example.

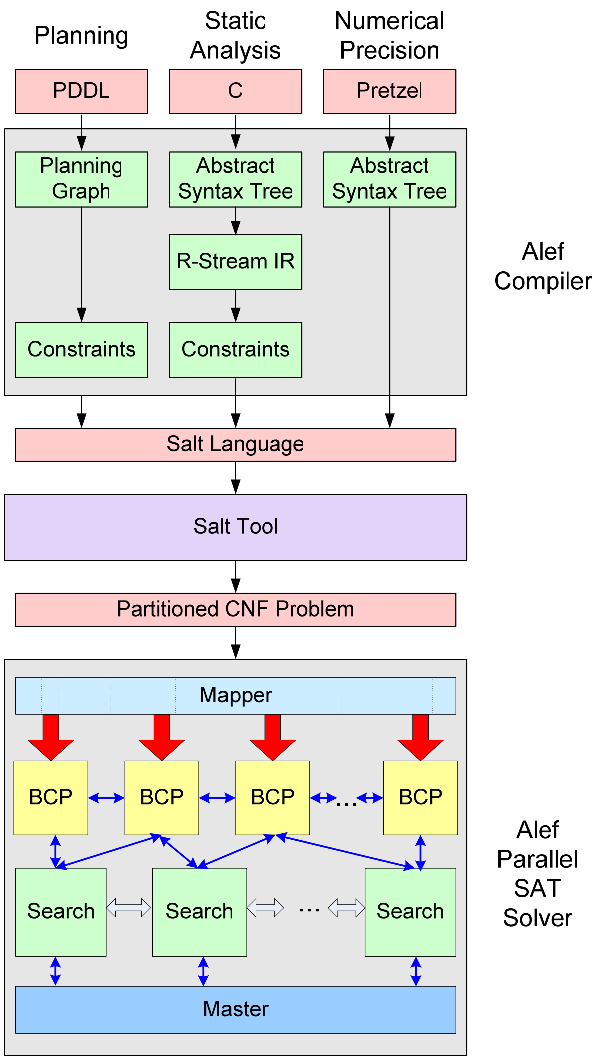
Design Choices

- **Distribute clauses over the processors of the machine**
- **Reduce bisection load on the network**
  - Message-driven implementation
  - Emphasize static mapping of clauses to improve locality
  - “GUPS problem”
- **Build competitive sequential solver first**

Alef System

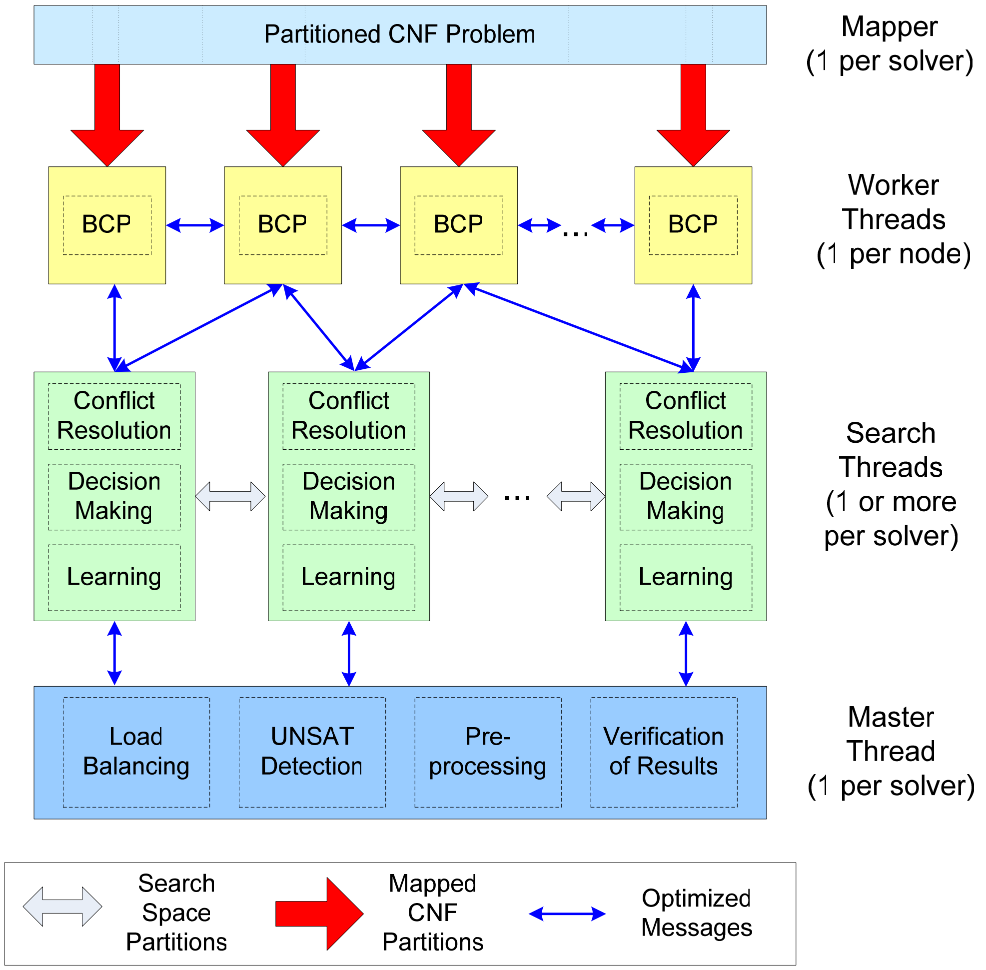
Alef system: uses Reservoir's R-Stream compiler technology, parallel SAT engine, and HPCS hardware to solve military and commercial planning and verification problems.

- **Alef compiler:** accepts planning and formal verification problems and transforms them to the Salt language.
- **Salt tool:** translates Salt language into CNF with partition annotations. Performs optimizations based on lazy-inference that reduce the size of the resulting CNF representation.
- **Parallel SAT solver:** incorporates parallel algorithms and state-of-the-art solver heuristics to achieve significant speedup on some structured problems.



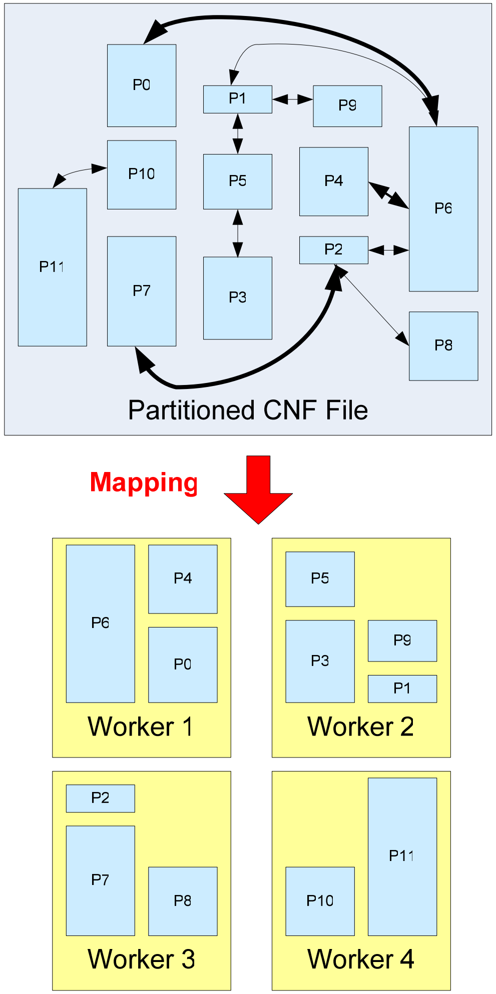
Alef Parallel SAT Solver

- Multithreaded implementation allows solver to explore the search space in parallel
- Message passing approach reduces network load and round-trip message delays
- Mapping algorithm distributes data to reduce communication load
- Solver supports decision heuristics tuned for specific problems
- Dynamic load balancing ensures processing resources remain busy
- Asynchronous sharing of learned information allows parallel threads to work together



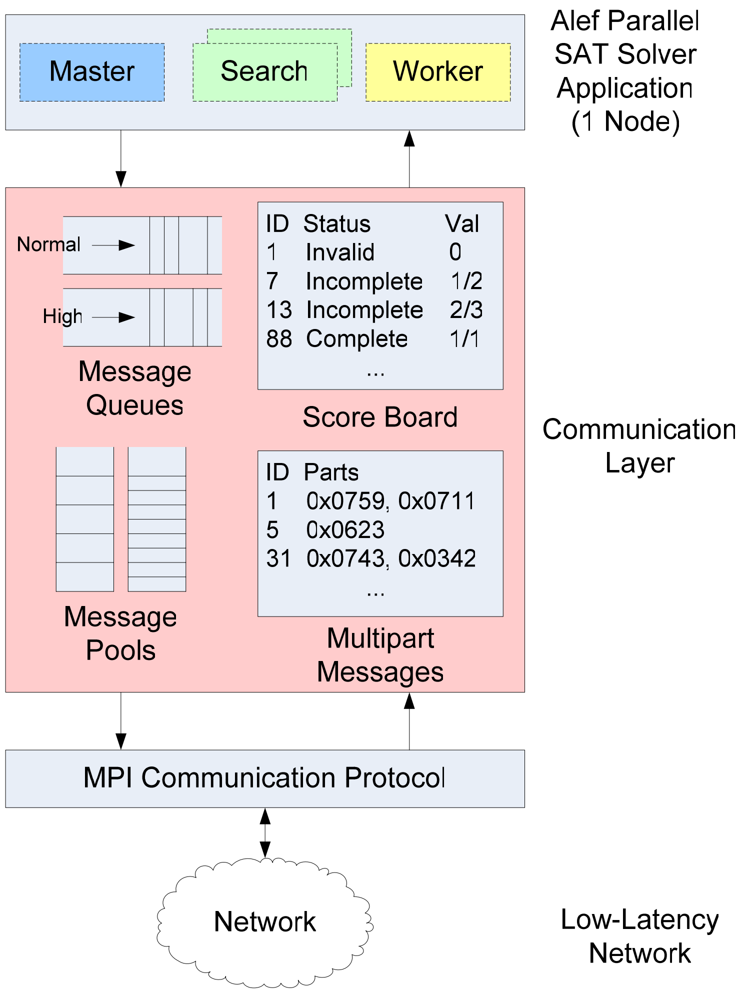
Alef Mapper

- **Mapping is the process of distributing clauses between worker threads**
- **Critical to performance because it determines the amount of communication required between nodes**
- **Places variable sized logical partitions of clauses into physical memory**
  - Optimal solution is NP-Hard; heuristic used instead
- **Mapper features:**
  - Places strongly connected logical partitions together
  - Balances distribution and replication of clauses based on problem size and communication overhead
  - Written for scalability and to allow for empirical tuning
- **Falls back to a pre-sorted first-fit greedy bin packing algorithm if mapping algorithm fails**

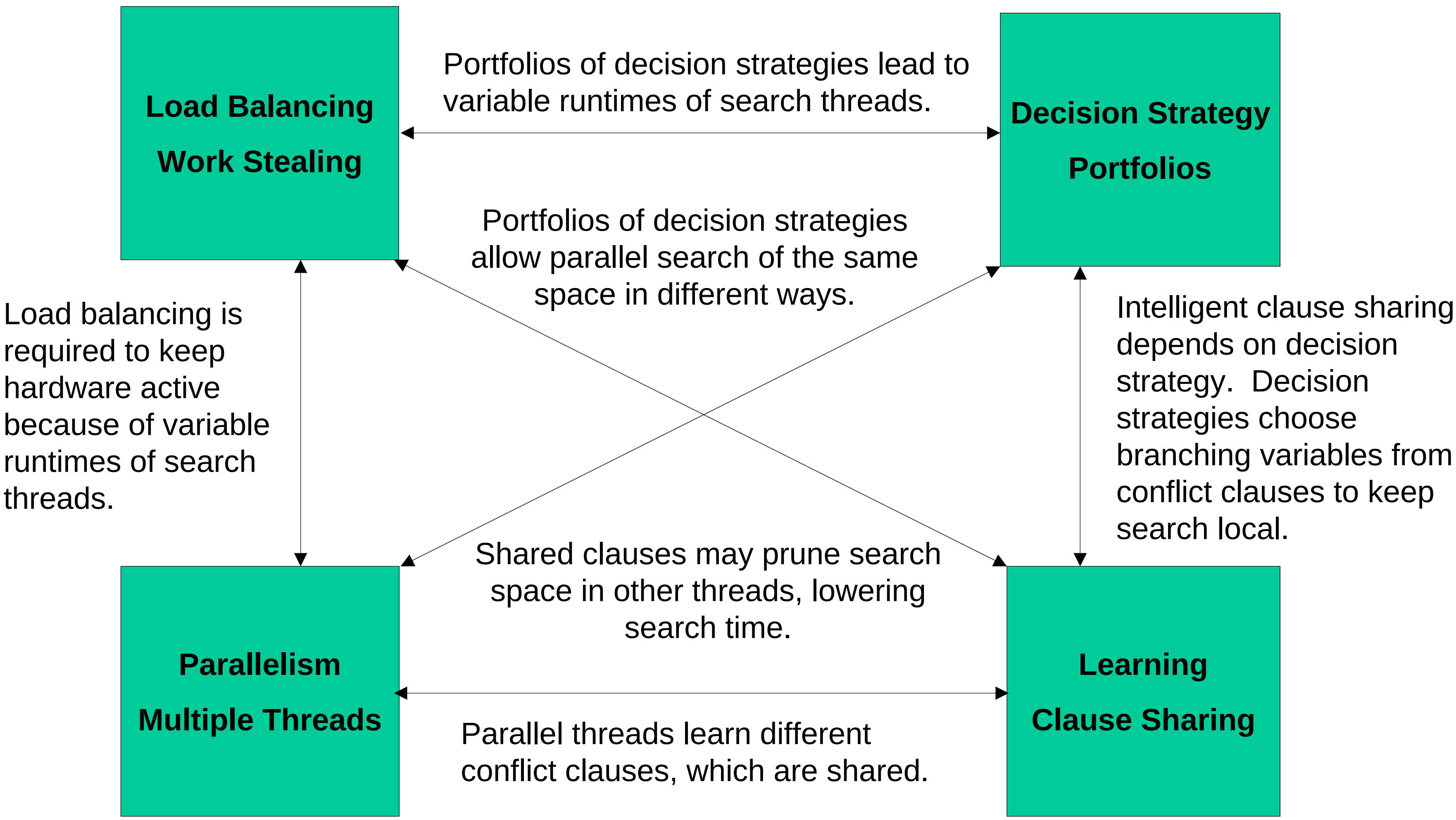


Communication Layer

- **Currently implemented on top of MPI**
  - Thin, low-overhead abstraction layer
  - MPI easily swapped out
- **Efficient**
  - Message pooling decreases overhead of MPI
  - Zero-copy inner loops
  - Message sizes optimized for machine
  - Space-sensitive encoding schemes
  - Local messages bypass MPI
- **Useful Semantics**
  - Provides one-sided send and receive, request and respond
  - Detection and elimination of stale messages
  - Transparent multipart message handling
  - High and normal priority queues

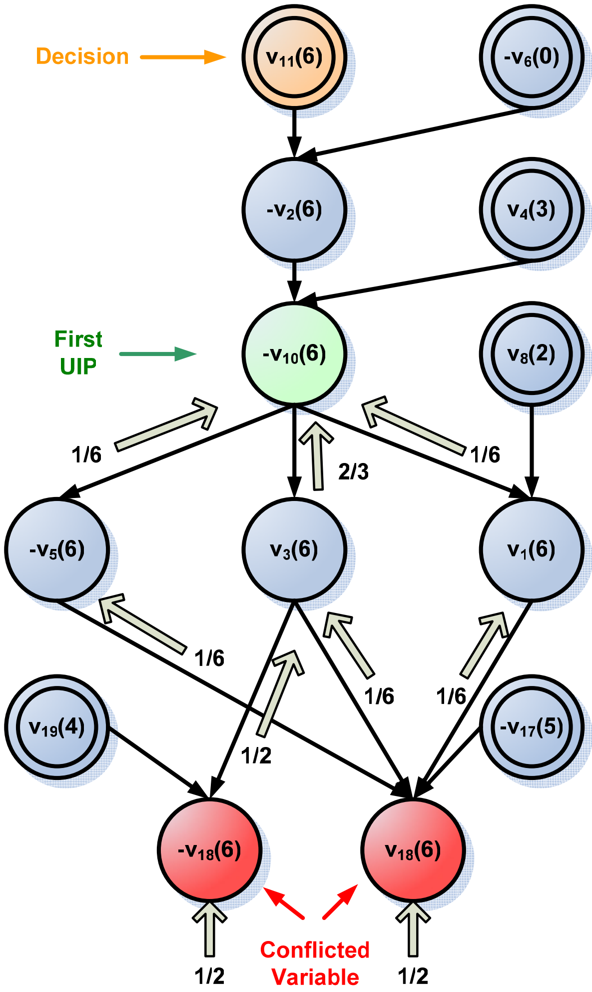


Parallel Heuristic Interactions



How? Distributed Conflict Resolution

- **Fully-distributed algorithm builds conflict clause**
  - Finds the articulation point between conflicted variable and decision variable closest to the conflict (first UIP)
  - First UIP is vertex where flow from conflicted variables sums to 1
- **Differs from distributed min-cut max-flow algorithm**
  - Not an optimization problem
  - Finds max flow between 3 vertices
- **Does not require construction of implication graph**
  - Minimizes storage requirements
- **Implementation is challenging**
  - Sidelined for later version of the solver



Simple Implication Graph

Sequential Conflict Resolution

- **Conflict resolution is a critical modern SAT solver heuristic**
  - Solver finds the combination of decisions that led to a conflict and *learns* by building a conflict clause
  - Conflict clauses prune the search space, making large problems tractable
- **First unique implication point (UIP) algorithm**
  - Solver walks backwards in directed acyclic graph of implications (implication graph) from the conflicted variables towards the decision
  - Finds the first articulation point between the conflicted variables and the decision point (the first UIP)
- **Implications are discovered in parallel but traversed sequentially**
  - Maintaining order for sequential traversal in a parallel environment is difficult
  - Implications arrive in any order and represent multiple implication graphs
- **Techniques for implementation in parallel environment**
  - Order marks are used to maintain correct order and distinguish between graphs
  - Other marks indicate the subset of unique copies of implications

How? Regulation of Message Driven Programs

- **Message-driven approach reduces round trip transactions on the network.**
- **But: what if some processor is a hotspot? How is backpressure induced, and what is the effect?**
- **Distributed cached shared memory approach "naturally" regulates computation, "smoothes" hotspots.**
- **Current approach: use memory for queues, but needs more work, experimentation, and thought.**

Current Status

- **Parallel solver on-line**
  - Multiple worker threads (BCP engines) on-line
  - Sequential implementation of conflict resolution
  - Optimized communication layer
  - Multiple decision heuristics
  - Running on Cray XD-1 and Xeon machines
  - *Signs of good scalability to 10 processors*

- **Salt language tool version 1.0 in distribution**
  - Provides a means for generating SAT from different problem domains (e.g., planning, software verification)
  - Annotations to assist mapper

- **Misc. Front ends implemented**
  - Generate SAT problem instances from C programs

- **An apparently simple problem offers some significant challenges to parallelization**