Parallelizing a SAT Solver with Hyperobjects

I-Ting Angelina Lee  Justin Zhang

March 18, 2010
6.884 Term Project Proposal
What Is A SAT Solver?

\( f \): Boolean formula written in Conjunctive Normal Form (CNF)

e.g. \((x_1 \vee x_2 \vee \overline{x_3}) \land (y_1 \vee y_2)\)

**Variables:** \(x_1, x_2, x_3, y_1, y_2\)

**Literals:** a variable or negation of a variable

**Clauses:** Literals OR-ed together

A SAT Solver solves the Boolean satisfiability (SAT) problem.
Why Parallelize a SAT Solver?

- Many useful applications exist:
  - program verification, electronic design automation, constraint solving in AI
    - SAT is NP-complete [Cook71]
    - Large search space: n variable $\Rightarrow 2^n$ possible assignments
    - Efficient implementation exists, some are even parallel [Chaff, MiniSat, GRASP, SATO, ManySat, PaMiraXT]
  - Give us insights to what linguistic mechanisms are useful for parallelizing large applications (hopefully).
    - various kinds of hyperobjects: reducers, holders, and splitters.
A Serial SAT Solver* Overview

At each recursion level:

```
constraint_propagate();
if no conflict then
  if all variables assigned then
    return SAT;
  else
    decide();
else
  analyze();
  if at top level then
    return UNSAT;
  else
    backtrack();
```

- **propagate unit clause**
- **conflict**: UNSAT clause exists
- **pick a new variable to assign value**
- **analyze conflict**: add a *learnt conflict clause*
- **undo assignments until conflict is resolved**

* Based on MiniSat
Data Structures Used

- **trail**: a log of variable assignments (in their assigning order)
- **learnt**: the learnt conflict clauses
- **assign**: the current assignment of variables, the reason for each assignment, and the “level” at which each variable is assigned.
- **order**: order of variables sorted by a heuristic “activity” value
- **stats**: statistics on the search, such as the number of decisions and inference propagations.
Goal #1: Parallelize a SAT Solver

- **Parallelization**: explore both assignments to a given variable in parallel
- **trail**: store on the frame
- **learnt**: shared data structure protected with locks
- **assign**: splitter – values must be restored when backtrack
- **order**: worker-local copies, may effect the quality of ordering but not the correctness of the program.
- **stats**: reducer – each worker can accumulate stats independently and combine later
Goal #1: Parallelize a SAT Solver

- **Parallelization**: explore both assignments to a given variable in parallel
- **trail**: store on the frame
- **learnt**: shared data structure protected with locks
- **assign**: splitter – values must be restored when backtrack
- **order**: worker-local copies, may effect the quality of ordering but not the correctness of the program.
- **stats**: reducer – each worker can accumulate stats independently and combine later

Need better understanding of the use and implementation of a splitter.
Goal #2: Enable Parallelized SAT Solver In Cilk-M

- Understand splitter better
- Implement hyperobjects necessary in Cilk-M
- Port the parallelized SAT solver to Cilk-M
Plan Of Execution

- Parallelize MiniSat (or something like it) in MIT Cilk-5.
  - a crude implementation for splitters exists
  - manually simulate reducer, or forfeit the stats data structure all together.
- Implement hyperobjects in Cilk-M
  - focus on the implementation for splitters
- Port the parallelized SAT solver to Cilk-M
Proposal of a Stencil Transformer – a Prototype of Cache-Oblivious Stencil Compiler in Cilk++

6.884 – Term Project
Yuan Tang, Steven Bartel
Background

- Each point in an n-dimensional spatial grid at time $t$ is updated as a function of neighboring grid points at time $t-1$, ..., $t-k$.
- Iterative PDE solvers – Multi-grid, Finite Difference Methods
- Cache misses $\rightarrow$ cache oblivious algorithm
Goals

• Based on Cilk++ technology, we will develop a stencil computation that accomplishes the following:
  – Faster
  – Generalized to n-dimensional space
  – Deals with boundary conditions
  – Irregular shaped boundary conditions
Speed

- Coarsen the base case
- Tiling or circular queue for more parallelism
- Tuning techniques
  - Register blocking
  - Loop unrolling
  - Cache by-passing instruction
  - more...
N-Dimensions

• Iterative spacial dimension cuts → time dimension cuts
Boundary Conditions

- Code manipulation (template meta-programming)
- Hardwire periodic / non-periodic boundary information into output code
Irregular Shapes

- More research (applications & implementation)
- Support polygon functions to mark boundary
- Apply template meta-programming
Input / Output

- **Input:** specification written in meta-language
  - Kernel: offset, timestep, coefficient
  - Generator: dimensions, dimension sizes, unit point, periodic / non-periodic, kernelInput: specification written in meta-language
- **Output:** c++ program
  - Specialized
  - To be run many times on different data
Measuring Success

- Speed: our transformer against lab3 cache oblivious algorithm
- N-Dimensions: compare to 1-dimensional code
- Boundary: periodic vs. non-periodic
- Irregular Shapes
  - Compare to original grid
  - Perhaps integrate into real application
- Term paper (6-10 pages conference paper)
Plan A

- Optimized code for 3 different stencils
  - Heat equation
    - 1D 3-point
    - 2D 5-point
    - 3D 7-point
  - Gauss-Seidel? Gauss-Seidel Red-Black?
  - Others?
    - Explicit? Implicit? Hybrid?
    - Lattice-Boltzmann?
    - Life game?
Plan B

• Generalized Transformer
  – Metaprogramming
  – Arbitrary N-dimensions
  – Periodic/Nonperiodic Boundaries
  – Irregular shapes
Timeline

• Current Progress
  – papers, contacts

• April 13, 15
  – Significant progress on 2 of 3 stencils
  – investigation of meta-programming applied to n-dimensions / boundary conditions

• End of term
  – 3 of 3 stencils
  – Term paper
  – As much of plan B as possible
Parallel Maximum Flow

\[ G = (V, E); c(u, v); f(u, v); |f| \]
Applications

- Resource allocation, scheduling, linear programming problems, graph problems (max bipartite matching)

Algorithms

- Augmenting Paths (Ford and Fulkerson, Edmonds-Karp, Dinitz)
- Preflow-Push (Goldberg and Tarjan)
Push-Relabel Algorithm

- Preflow -- allow excess flow at a vertex
- Assign a height value to each vertex; \( \text{height}(s) = |V| \) and \( \text{height}(t) = 0 \)

- \textbf{Push}(u,v): send part of the excess flow from \( u \) to \( v \)
  - \( \text{excess}(u) > 0 \); \( c(u,v) - f(u,v) > 0 \); \( \text{height}(u) > \text{height}(v) \)

- \textbf{Relabel}(u): increase \( \text{height}(u) \) s.t. for at least one vertex \( v \)
  with \( c(u,v) - f(u,v) > 0 \) \( \text{height}(u) > \text{height}(v) \)
  - \( \text{excess}(u) > 0 \); \( \text{height}(u) \leq \text{height}(v) \) for all \( v \) s.t. \( c(u,v) - f(u,v) > 0 \)
Parallelization Approach

1. Apply *Parallel Breadth-First-Search* to Push-Relabel
   - *Global Relabeling* heuristic (based on BFS)
   - *Gap Relabeling* heuristic

2. Try parallelizing node *discharge* operations (Push-Relabel as long as there is excess)
   - e.g. using some locking scheme (node and neighbors)

3. Apply *P-BFS* to an augmenting path algorithm

4. Map max-flow to an interesting application and specialize for that problem
Parallelizing Single Source Shortest Paths

Kevin Kelley, TB Schardl

March 18, 2010
The Problem

- **Single Source Shortest Paths (SSSP)** Given a graph $G = (V, E)$ with non-negative edge weights, and a starting vertex $v_0$, find the shortest path from $v_0$ to every $v \in V$.

- **Solution 1** If all edge weights are identical, use breadth-first search (BFS). We know how to parallelize BFS efficiently (see Lab 4).

- **Solution 2** In general, we can use Dijkstra’s SSSP algorithm: Start with a minimum priority queue $P$ containing only $v_0$. Repeatedly remove the top element of $P$, $u$, and for each unvisited neighbor of $u$, $v$, set $\text{dist}.v = \text{dist}.u + w(u, v)$, where $w(u, v)$ is the weight associated with edge $(u, v)$. A good parallel version of this algorithm is not obvious.
Try parallelizing another SSSP algorithm, such as Gabow’s scaling algorithm.

Idea: Consider edge weights one bit at a time.

After each new bit is revealed, recompute edge weights and update shortest paths based on new information.

Each iteration divides the vertices into several buckets ordered by current weight, then re-optimizes the paths in successively larger buckets.

For more information on this algorithm, see CLRS 3rd edition, problem 24-4.
TODO (Plan B)

- Implement Gabow’s scaling algorithm.
- Implement Dijkstra’s SSSP algorithm, and compare the two serially.
- Parallelize Gabow’s scaling algorithm.
- Analyze performance, both theoretically and in practice.
TODO (Plan A)

- Compare to some parallel Dijkstra implementation.
- Investigate modifications of Gabow’s algorithm to improve performance.
LU AND QR FACTORIZATION IN CILK++

Nathan Beckmann
Silas Boyd-Wickizer
THE PROBLEM

- LU- and QR-decomposition are common matrix operations with a broad range of applications
  - LU: Solve systems of linear eqns, determinants
  - QR: Linear least-squares, eigenvalues

- Both factor a matrix into multiple matrices with useful properties
PLU Decomposition

- Decompose matrix $A$ into a pivoting matrix $P$ and a lower- and upper-triangular product, $LU$

$$PA = LU$$
**QR Decomposition**

- Decompose matrix $A$ into an orthogonal matrix $Q$ and an upper-triangular matrix $R$

$$A = QR$$

Where $q_{ij}$ are unit, orthogonal vectors.
**Issues**

- *We aren’t trying to invent a new parallel LU/QR algorithm!*

- LU-decomposition can be implemented using divide-and-conquer

- This gives asymptotically optimal cache behavior, but no parallelism
  - Branches of division are dependent

- Instead of parallelizing high-level algorithm, parallelize individual matrix operations (multiply, backsolve, etc.)

- Similar for QR
Success Metrics

- Unlikely to beat MKL, PLASMA, LAPACK, etc. on performance

- Instead, combine performance and code style
  - Raw performance, scaling, etc.
  - Lines of code, code complexity, development time

- Not clear how to combine these numerically – evaluate qualitatively
PLAN B

- Implement LU-decomposition in Cilk++
  - Have implementation for Sivan Toledo’s algorithm courtesy of Bradley
  - Parallelize standard, “right-looking” algorithm
  - Use linear algebra libraries for base case
  - Achieve ‘competitive’ performance

- Compare against highly-optimized LU implementation
  - Raw performance and scalability
  - Lines of code and coding style
**Plan A**

- Implement QR factorization in Cilk++ with competitive performance
- Understand performance
  - Impact of randomized scheduler vs. static scheduling
  - System factors – cache behavior, off-chip memory bandwidth, etc.
- Compare across different machine architectures
Speculative Execution and abort in Cilk++

Ruben Perez & Gregory Malecha

MIT

April 13, 2010
Problem?

- Lots of algorithms can be parallelized by speculative execution.
  - Considerable increase in the amount of work.
  - Need to be able to abort speculative computations when their results aren’t necessary.
- Previous cilk5 supported abort natively, but Cilk++ doesn’t.

Problem

How can abort be implemented as a user-level library and what are the performance trade-offs of doing this?
Native abort

```c
int first() {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn long_computation_1());
    reduce(cilk_spawn long_computation_2());
    cilk_sync;
    return x;
}
```
Native abort

```c
int first() {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn long_computation_1());
    reduce(cilk_spawn long_computation_2());
    cilk_sync;
    return x;
}
```

- No easy way to do this in Cilk++...
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  - Poll up to the root of the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  - Poll up to the root of the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  - Poll up to the root of the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  - Poll up to the root of the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  - Poll up to the root of the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree

```
  1
 / \
1   0
|   |
0   0
|   |
0   0
|   |
0   0
```
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
User-Implemented abort

- Can’t modify the runtime/scheduler to handle abort.
- Implement with polling...
- Two possible implementations:
  1. Poll up to the root of the tree
  2. When we abort we push the abort flag down the tree
Plan B

- **Implementation**
  - An easy-to-use interface (able to interact with existing code)?
  - Implement `abort` both ways.
  - Port Cilk Pousse to work on Cilk++.

- **Evaluation**
  - **How** Which one is faster?
  - **How often** Compare for algorithms with different amount of work in the body. How much work is saved?
Plan A

- **Implementation**
  - Port existing `cilk5` program to use our `abort`.
    - Implements heuristics on top of $\alpha$-$\beta$-pruning.
  - Reducers might be needed to replace inlets.

- **Evaluation**
  - Can we search deeper when we use `abort`?
  - Do we spend a lot of time polling?