Parallelizing k-SAT Solvers

Towards a SIMT algorithm

Vikrant Gajria — vgajria@purdue.edu — May 2022

What is k-SAT problem?

Overview of problem and approach

- **CNF:** Given clauses C such that each clause consists of at-most k propositional variables/ literals l joined by a logical OR operator $\vee$, and each clause is joined by a logical AND operator $\wedge$.
  Example: $(p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (\neg q)$
- **Problem:** Can we find a model/ interpretation of propositional variables that would satisfy all clauses of given input CNF expression?
  Example: $p \rightarrow \bot \quad r \rightarrow \top \quad q \rightarrow \bot$ (False, True, False) is one such interpretation, though the logical expression may have many more models.
- This problem was the first **NP-Complete** problem *(Backtracking!)*.

Deduce: Unit Propagation

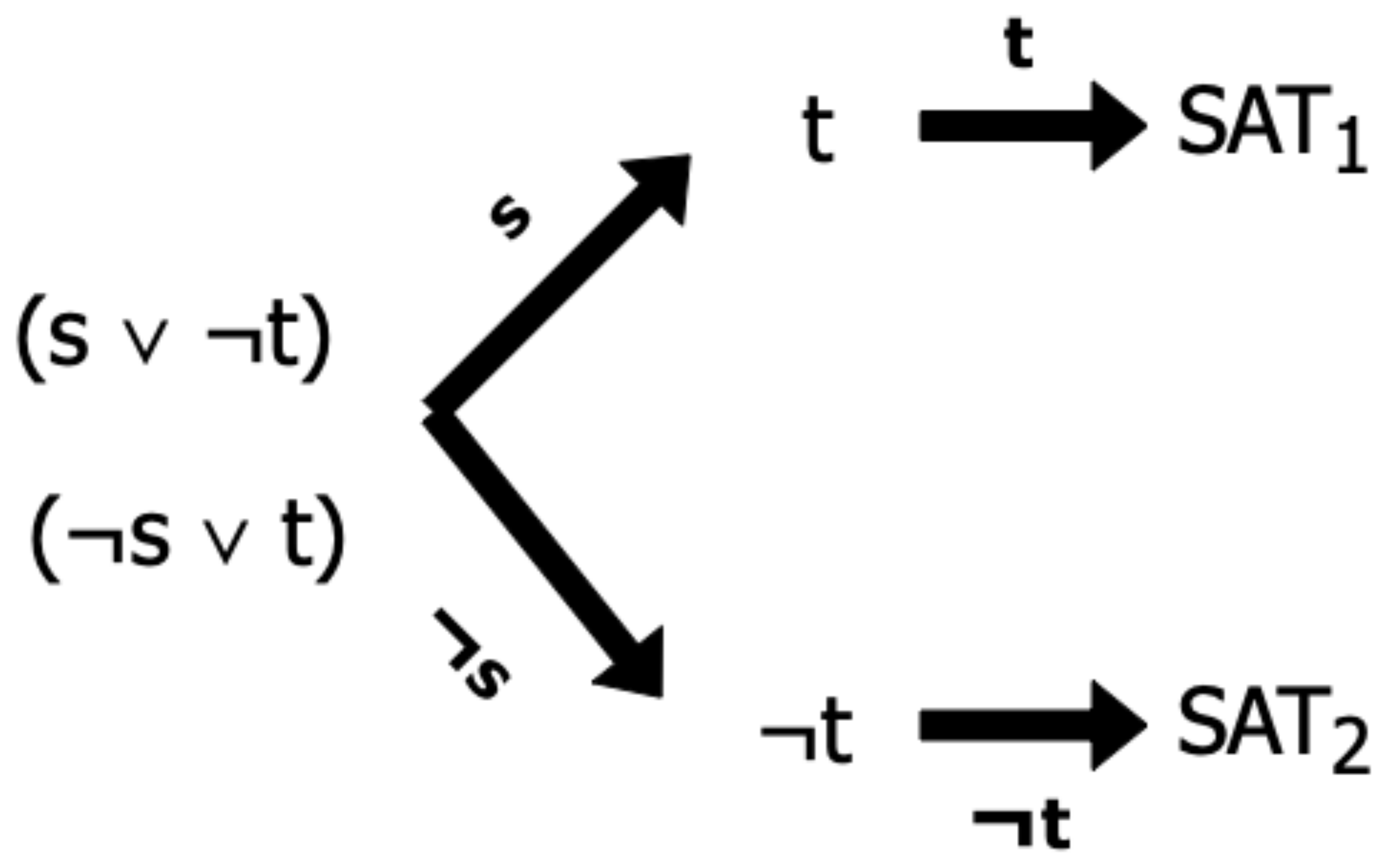
Powerful step to resolve a bulk of clauses!

- **Example:** $(p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (\neg q)$
- **Note:**
  - $(\neg q)$ is a clause of only one literal.
  - In CNF, we need to satisfy all clauses, therefore we cannot set $q \rightarrow \top$ because that would UNSAT the expression.
- **Conclusion:** We can ***deduce*** that any model that satisfies the given example ***must*** map $q \rightarrow \bot$.
- **Example becomes:** $(\neg p \vee r) \wedge (p \vee r)$

Guess: Backtracking

When you can't use optimization axioms

- **Example:** $(\neg s \vee t) \wedge (s \vee \neg t)$
- Since you can't deduce any literal using previous axioms, you need to ***guess*** and backtrack.
- Exploratory decomposition is trivial.
- Therefore, such decomposition is **not the focus of this project!**



DPLL Algorithm for k-SAT solving

60+ year old algorithm, still used today (on CPUs!)

Input: Set of clauses Φ. ***Output:*** Truth value indicating whether Φ is SAT.

while there is a **unit clause** $\{ l \}$ in Φ do

    $\Phi \leftarrow$ **unit-propagate**(l, Φ); *// Deduce*

Parallelize this step

if Φ is empty **then** return $\top$; *// No more clauses left to satisfy*

if Φ contains an empty clause **then** return $\perp$;

$l \leftarrow$ **choose-literal**(Φ);

return DPLL($\Phi \wedge \{ l \}$) or DPLL($\Phi \wedge \{\neg l\}$); *// Guess*

Terminology

- An **assignment** is settings a variable (like p in $(p \vee q)$) to True ⊤ or False ⊥
  - Each processor should try to find an assignment for the clauses that it has been assigned
- A model has **conflicting assignments** if for the same variable v, v is set to True as well as False
  - Processors working independently on clauses may come up with conflicting assignments, which must be reduced onto one processor (this is done using bit operations on root processor only, to reduce synchronization)
- A **clause** with at least one assigned variable, such if v appears in clause as well as in model with same polarity (+ or -), is said to be **SAT**isfied.
  - If multiple processors are working on the same clause, they must keep track of whether the clause has been satisfied somewhere or not, and this information must also be reduced ("if any processor finds SAT" = logical OR)
- If all clauses are satisfied, then **formula is SAT** algorithm must halt.
  - This information must be reduced from all processors working on various clauses ("if all processor finds SAT" = **logical AND**)
- A **roadblock** is found when a clause is not SAT, and there are no unassigned literals left in the clause.
  - If any processor hits a roadblock, then model does not SAT the formula. This information must be reduced from processors working on different clauses ("if all processor finds roadblock" = **logical OR**)

Abstract Data Representation

Instead of Sets, use a matrix representation

Refer to implementation details for a more efficient manner to represent this

$$(p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (\neg q)$$

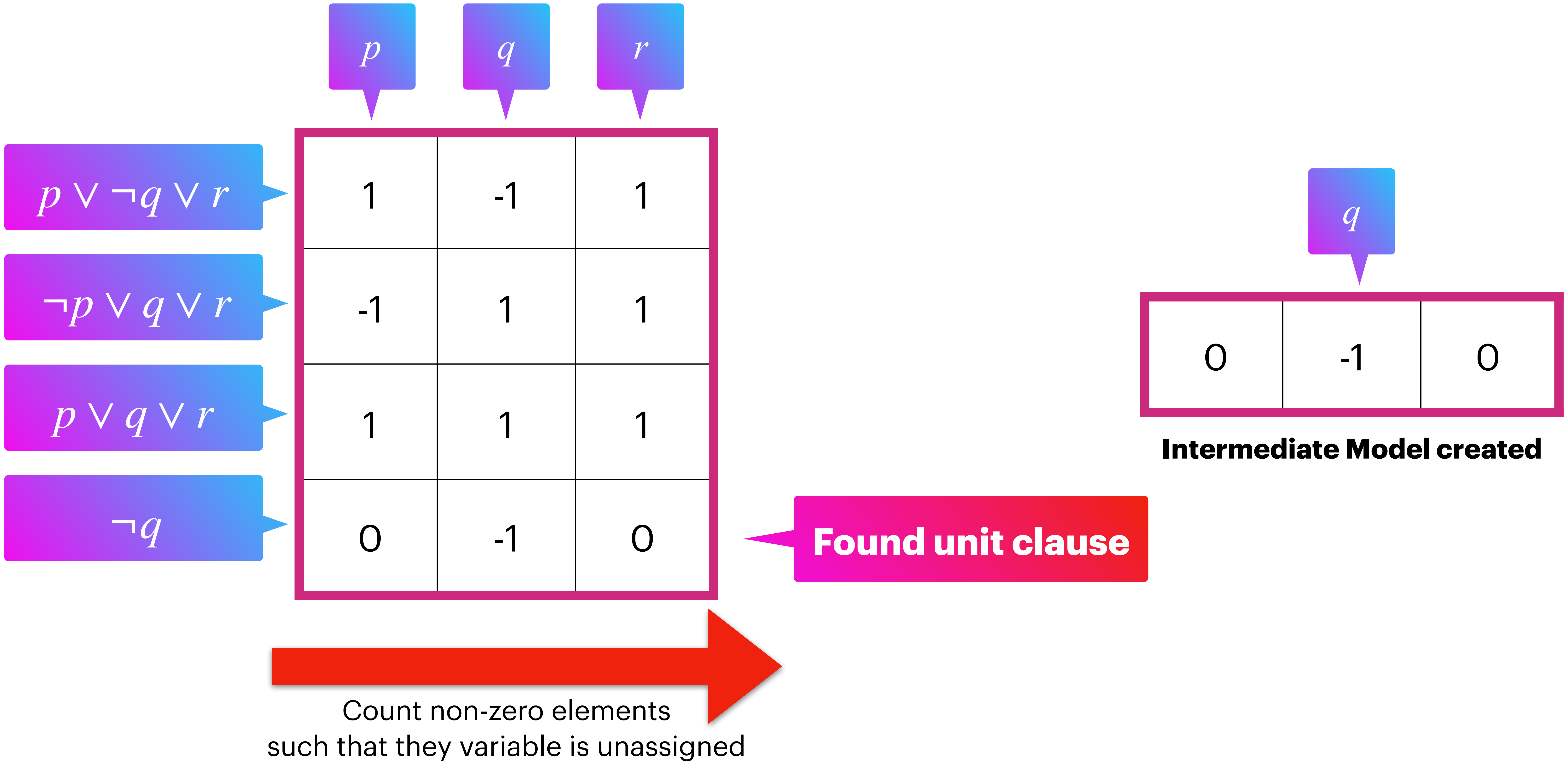
Convert to **2D Mesh** M_{ij}

- Every row represents a clause C_i
- Every col represents a literal l_j
- $M_{ij} = 1$ if l_j appears in C_i without negation
- $M_{ij} = -1$ if l_j appears in C_i with negation
- $M_{ij} = 0$ if l_j dose not appear in C_i

	p	q	r
$p \vee \neg q \vee r$	1	-1	1
$\neg p \vee q \vee r$	-1	1	1
$p \vee q \vee r$	1	1	1
$\neg q$	0	-1	0

Naive Unassigned Literal Search

In order to prove that a clause is not a unit clause, a linear scan is usually performed to find unassigned variables



Optimal Unit Propagation: Watched Literals

Chaff: Engineering an Efficient SAT Solver, DAC 2001

- Lazy data structure for determining if a clause is unit clause
- If you know that a clause has **two unassigned literals**, then the clause is not a unit clause
- Only when you assign a **watched literal** to False, then **find** another literal to prove that the clause is not a unit clause
- If another unique unassigned literal cannot be found, then the clause is a unit clause and you already know the literal that can be deduced!

$\boxed{p} \lor \boxed{q} \lor r$

- Assign p to false
- p was a watched literal
- Therefore, find a new literal to watch

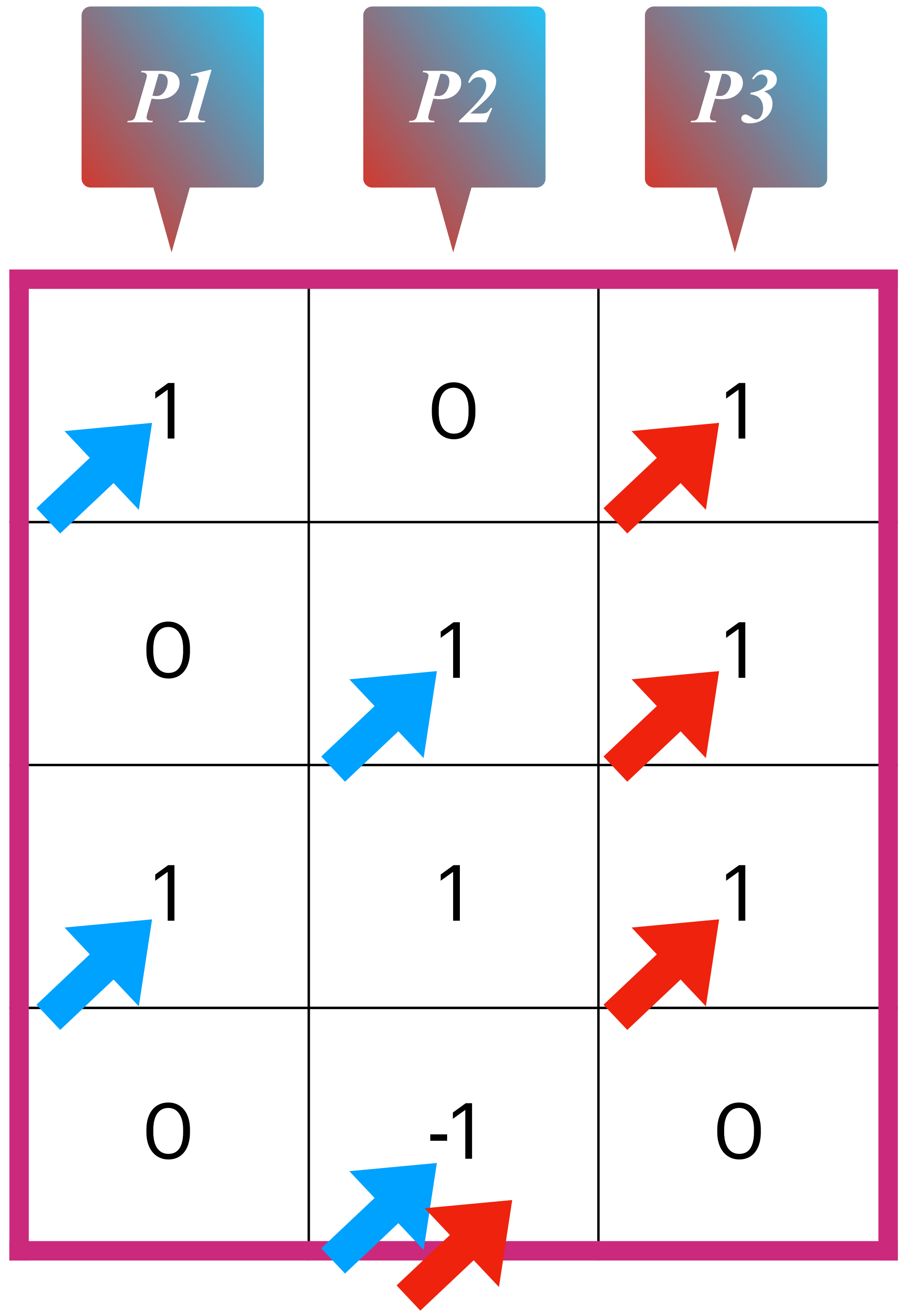
$\lor \boxed{q} \lor \boxed{r}$

- Backtrack p to false
- Now, this is a lazy datastructure
- Since we know that r is unassigned, no need to update watched literal information!

$p \lor \boxed{q} \lor \boxed{r}$

- If you assign some literal s that is not watched then watch pointers are not updated!

Column decomposition: Min-Max Watched Literals



Since min-index == max-index,
This is a unit clause

It only has $\neg q$

Therefore we set $q \rightarrow$ False

- For every clause, find **unassigned literals that have minimum and maximum indices in the row**
- This can be found using **parallel reduction**
- Reduces literal search from $O(l)$ to $\Theta(\frac{l}{p} + \log_2 p)$ where l is number of literals in a clause.
- **Isoefficiency function** is $\Theta(p \log_2 p)$
- Assuming each processor has $\frac{l}{p}$ literals that it can search through to find minimum and maximum indices of unassigned literals
- While searching for unassigned literals, also **check if all clauses are satisfied**

Row decomposition: Trivial Work Distribution

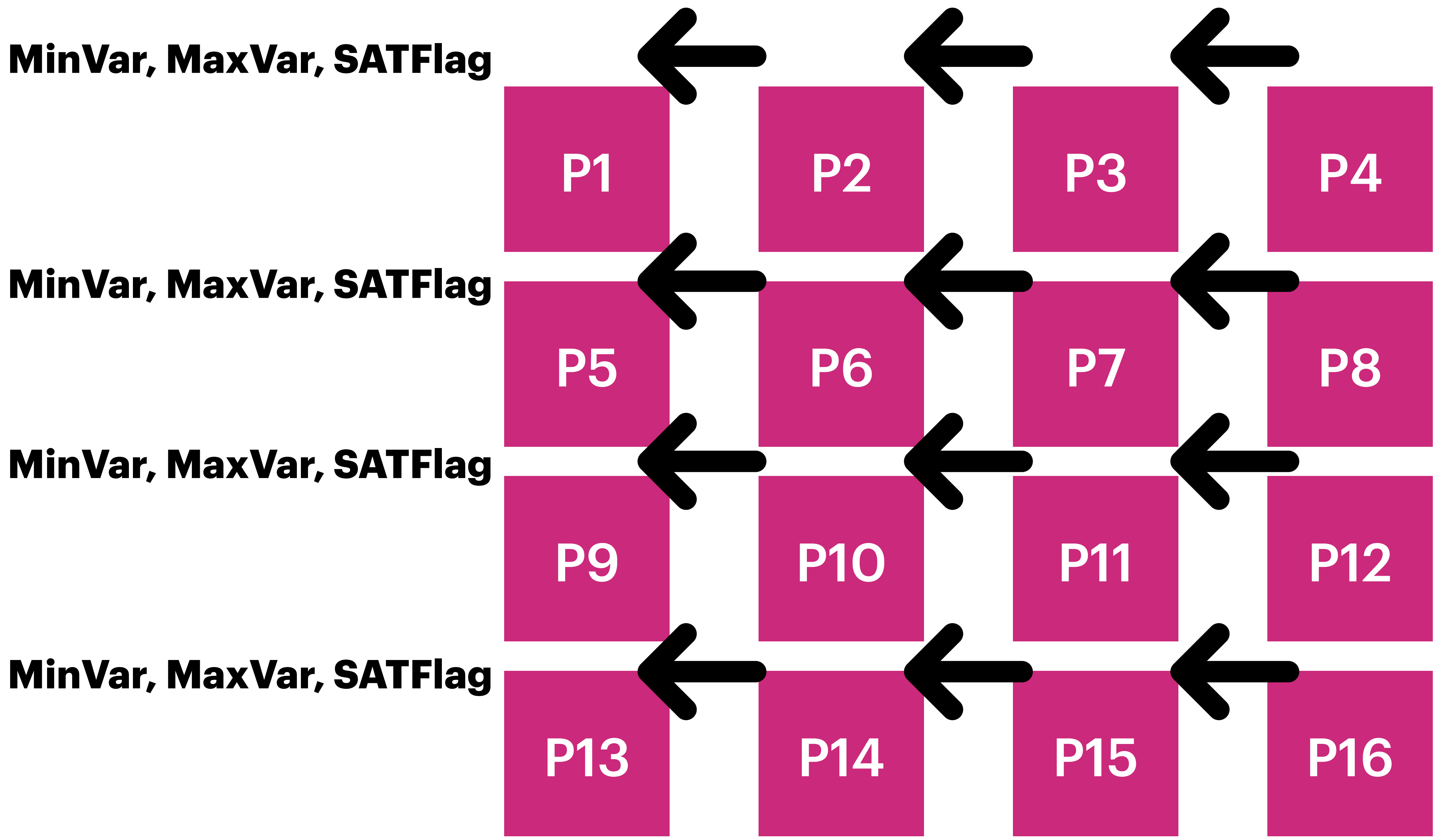
P1	1	0	1
P1	0	1	1
P2	1	1	1
P2	0	-1	0

- Finding unit clauses is an independant task
- Every processor is assigned $\Theta(\frac{c}{p})$ clauses
- Once every processor is done with finding unit clauses, it creates an **intermediate model** of size $\Theta(l)$
- Serial work is $\Theta(cl)$
- Intermediate models are then **reduced onto one processor** in time $\Theta(\frac{c}{p}l + pl)$

Example of algorithm, using both row and column decomposition

Each processor is assigned $\frac{l}{\sqrt{p}}$ literals of $\frac{c}{\sqrt{p}}$ clauses to search from. Broadcast, in MPI, takes $O(cl\log_2 p)$ time.

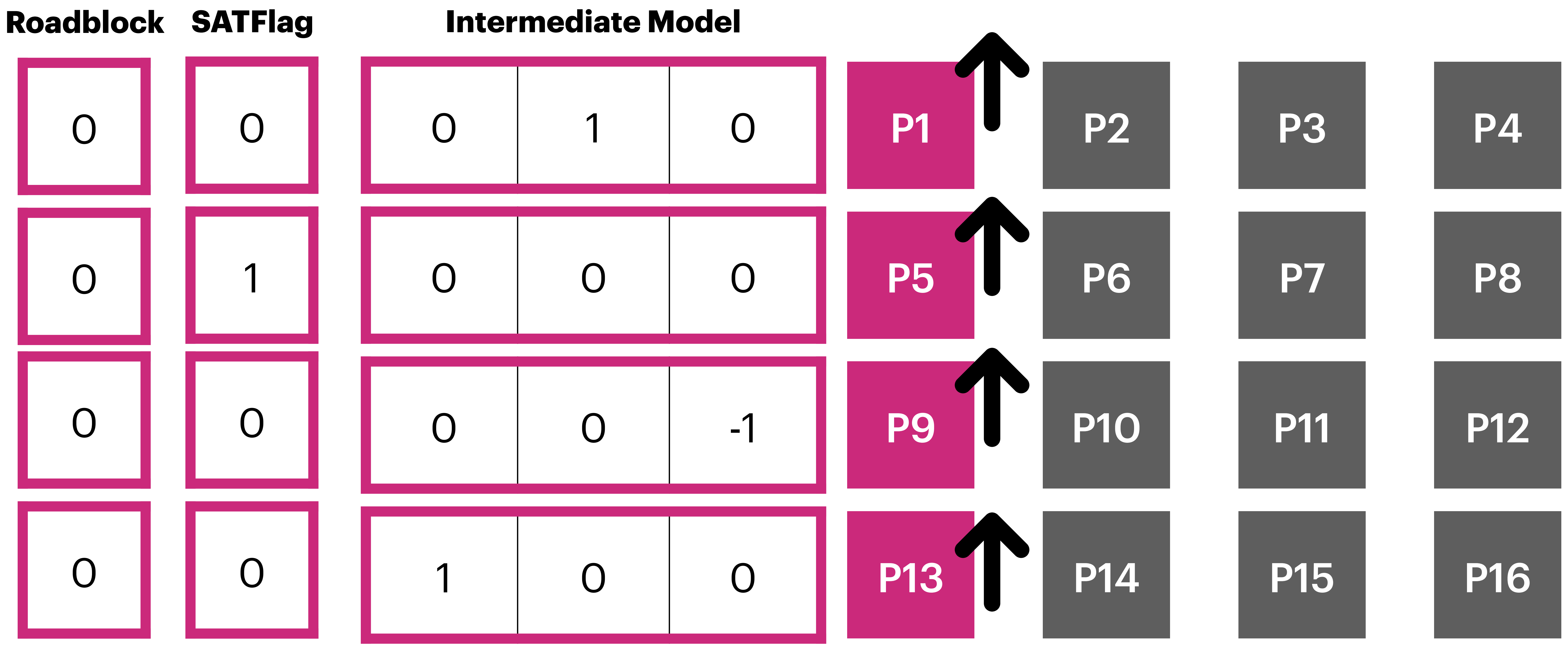
Inner Loop: Finding Unit Clauses and SAT Clauses




```
#pragma omp parallel \
    for \
    if(COL_STRIP == 1) \
    shared(i, m) \
    reduction(||: sat) \
    reduction(min: minv) \
    reduction(max: maxv)
for (int v = 0; v < i.numv; v++)…
```

- Each processor is searches through $\frac{l}{\sqrt{p}}$ literals to find unassigned literals, and only keeps track of minimum and maximum index of unassigned literals. If an assignment to a literal satisfies the clause, then a boolean SAT flag is set to True.
- Processors in a row then perform row-wise reduction using min, max, and logical OR operation to accumulate the result for each clause.

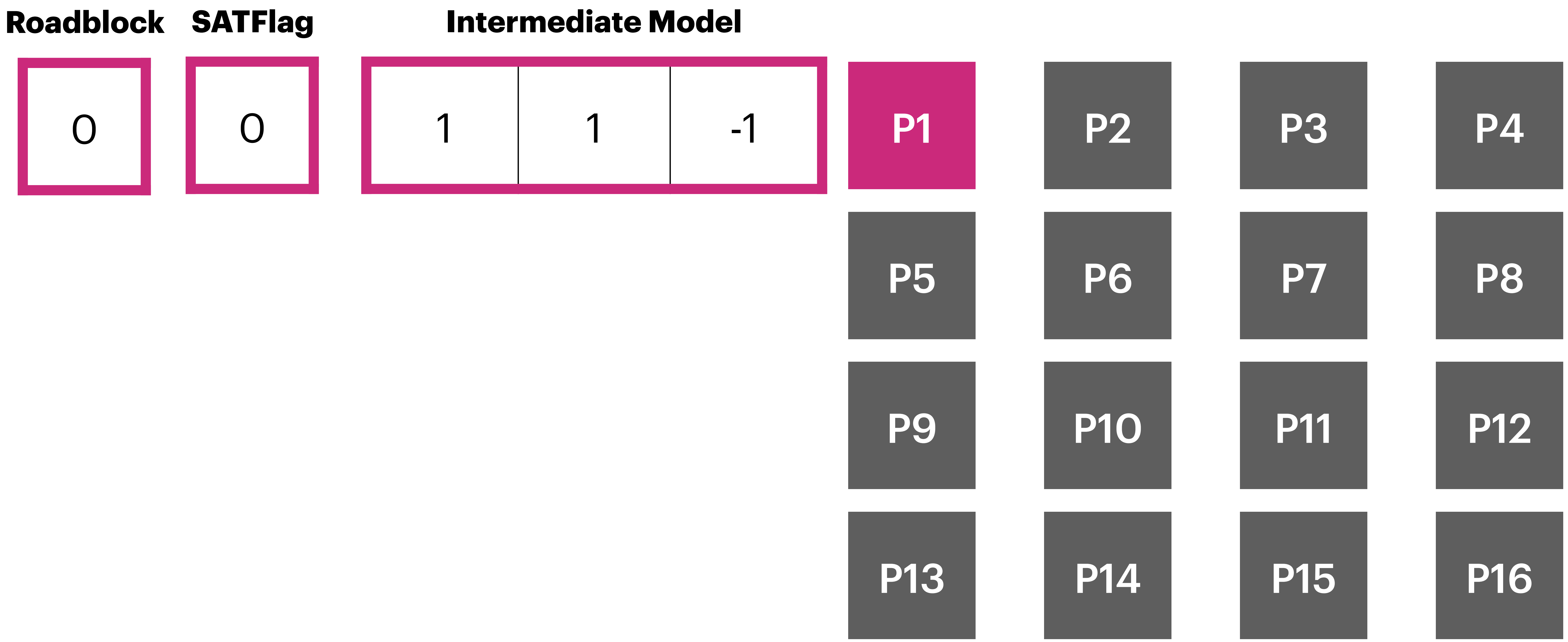
Outer loop: Reducing Intermediate Models




```
#pragma omp parallel \
    for \
    if(ROW_STRIP == 1) \
    shared(i, m, change) \
    reduction(&&: allsat) \
    reduction(||: roadblock)
for (int c = 0; c < i.numc; c++)…
```

- Every clause may result into at-most one assignment (since it is a unique clause)
- These assignments are represented has l length arrays and are combined together using another reduction operation (Refer implementation details on how to reduce!)
- All processors in one column also contain information on whether their row has been satisfied. If all processors say that their clause is satisfied, then the formula has been satisfied and the algorithm need not proceed further. Furthermore, the processors can also judge if a clause cannot be satisfied, and this information is also reduced onto one root processor.

Broker thread: Checking for Conflicts and Roadblocks



- If the intermediate model found after this procedure is different as compared to the previous model, then the procedure is repeated!
- If a conflict in the model, or a road-block (UNSAT clause) is found, then this branch of state-space search is not on the right path, and we need to backtrack.
- This is an NP-complete problem, therefore the algorithm takes a long time to execute and repeats steps indeterministic number of times.

Implementation Details: Bitmasking

- A formula matrix Φ of c clauses and l literals is represent as **2 bit sets** of lengths $c \times l$. The first bit set represents occurrence of **positive polarities** of literals in clauses, and the second bit set represents occurrences of **negative polarities**.
- Similarly, an **intermediate model** M is represented as **positive/ negative bit sets** of lengths l for each literal.
- A model is said to contain a **conflict** if a **literal is marked in both positive and negative bit sets** — that is, $\text{pos}(M) \wedge \text{neg}(M) \neq 0$.
- Using bit masks helps in **reduce memory requirement** by storing **information of 4 literals in 1 byte of data** instead of 1 byte per literal. This further helps in cache hit ratio.

Problem with Column Decomposition

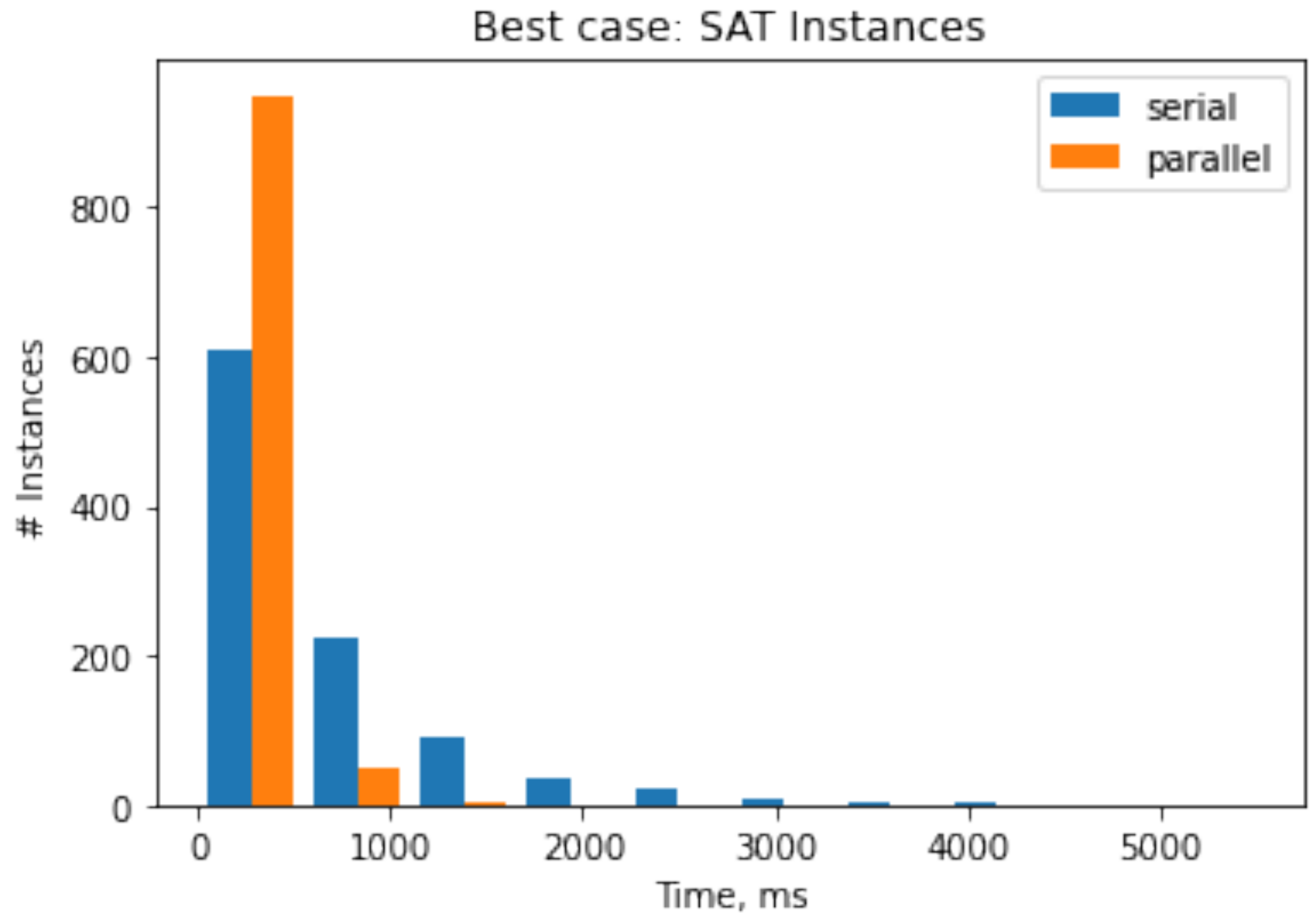
- On experimentation, I found that **communication overhead of column stripped decomposition drastically slows down the performance** of the algorithm. Test cases just wouldn't finish in 24 hours.
- In the datasets that I found $l << c$, it didn't make sense to enable min-max search due to communication overhead. An entire row only takes up **at-most 16 uint32_t** when using bit sets, which is amazing small and thus has no need for parallel search. Linear search works faster here, and it we use bit sets, its easy to find LSB and MSB of bit sets in a fast manner. You can also check if a clause is a unit clause using: https://stackoverflow.com/questions/12483843/test-if-a-bitboard-have-only-one-bit-set-to-1
- By $T_p = \Theta(c\frac{l}{p} + \log_2 p)$, setting the derivative of $T_p' = 0$, we get $l = \frac{p}{c}$. For very large c, column decomposition starts negatively impacting performance.
- Therefore, **column division has been disabled in the experimentation** and only row division is used.
- However, column division can be enabled for when number of columns $\approx$ number of rows.

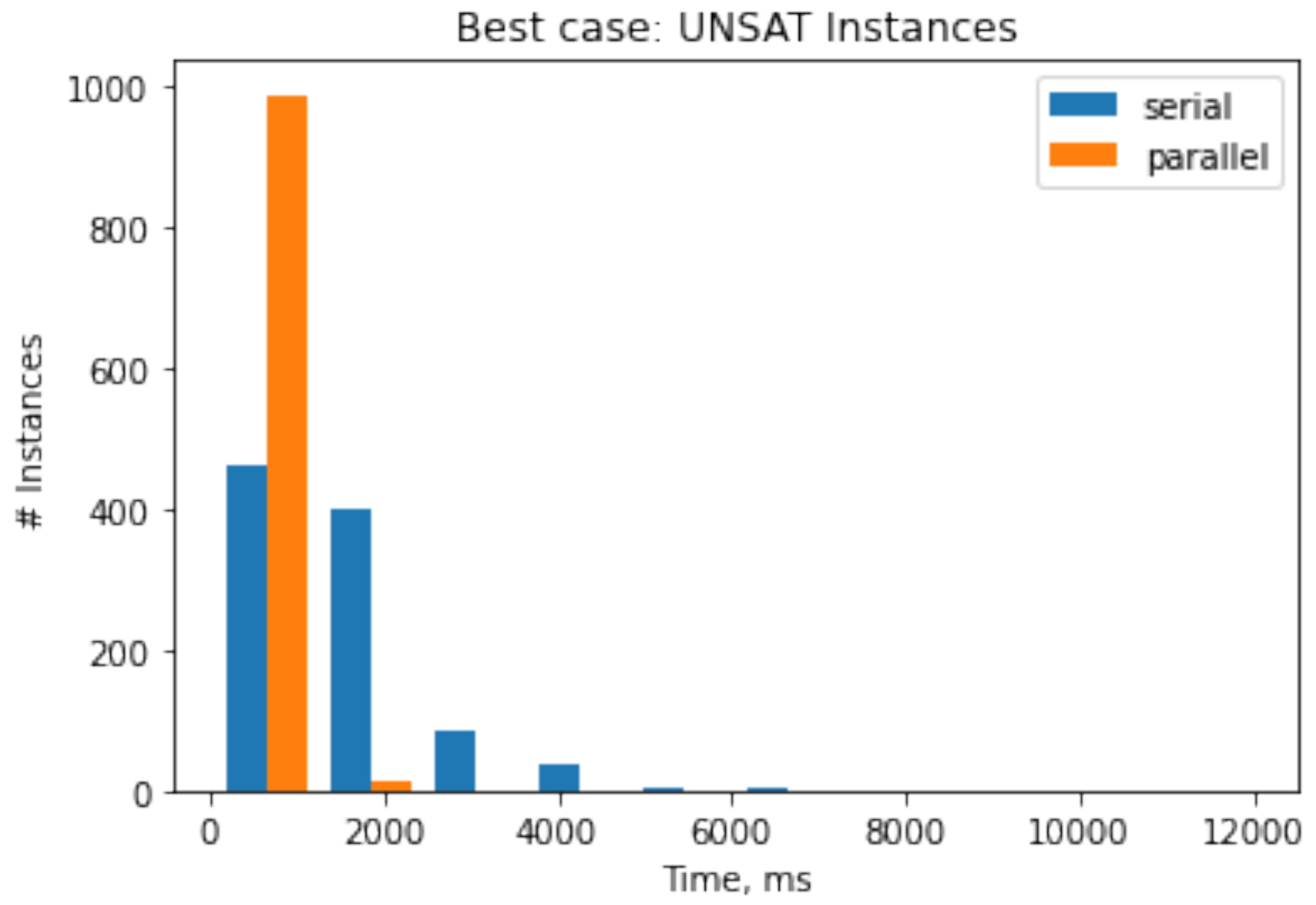
Evaluation Technique

- Tests run on mc17, mc18, mc19 and mc20 Purdue servers with 24 and 32 threads respectively
- SAT cnf files were used for varied problem size
- Row decomposition was enabled, column decomposition was disabled due to small problem sizes
- Source: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

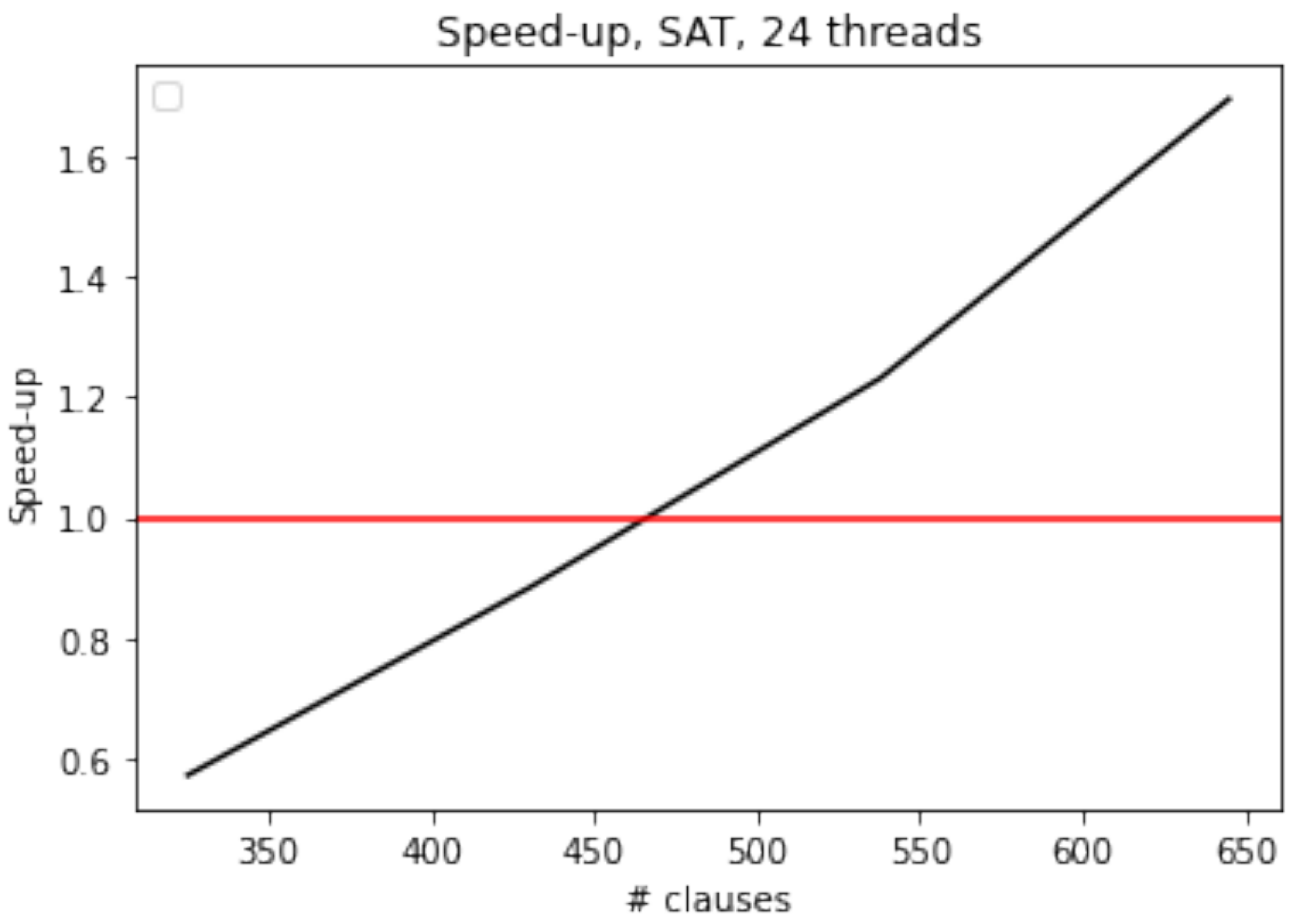
Results, $l = 50 \quad c = 218$

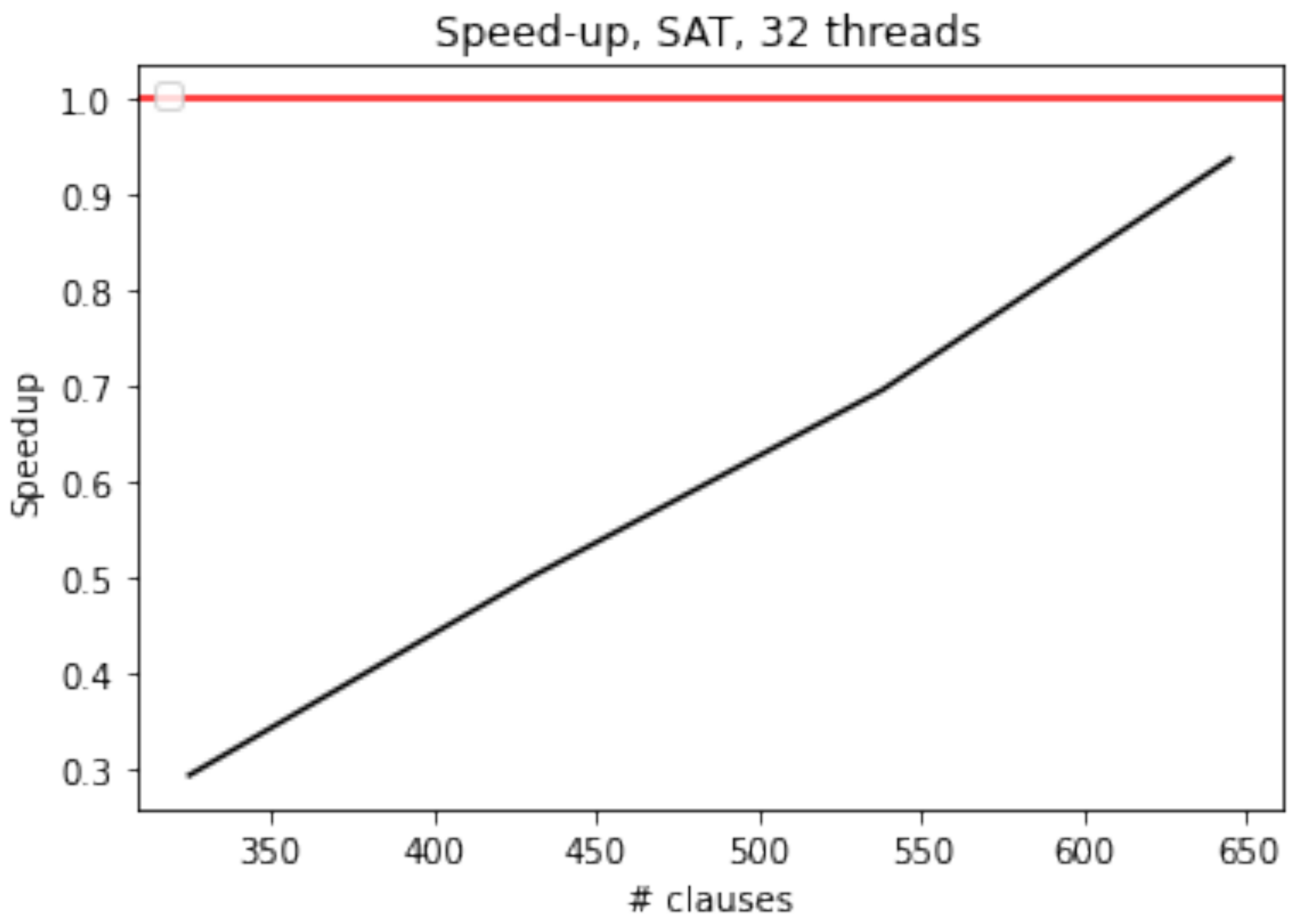
- Histograms show # of instances in time brackets.
- There are **more instances finishing before 1000 ms with parallel algorithm** as compared to serial algorithm in SAT instances.
- This is more prominent in UNSAT instances where almost all of the state-space is searched.
- In both best (SAT) and worst (UNSAT) cases, parallel algorithm with 32 threads is on average **72% faster** than serial algorithm



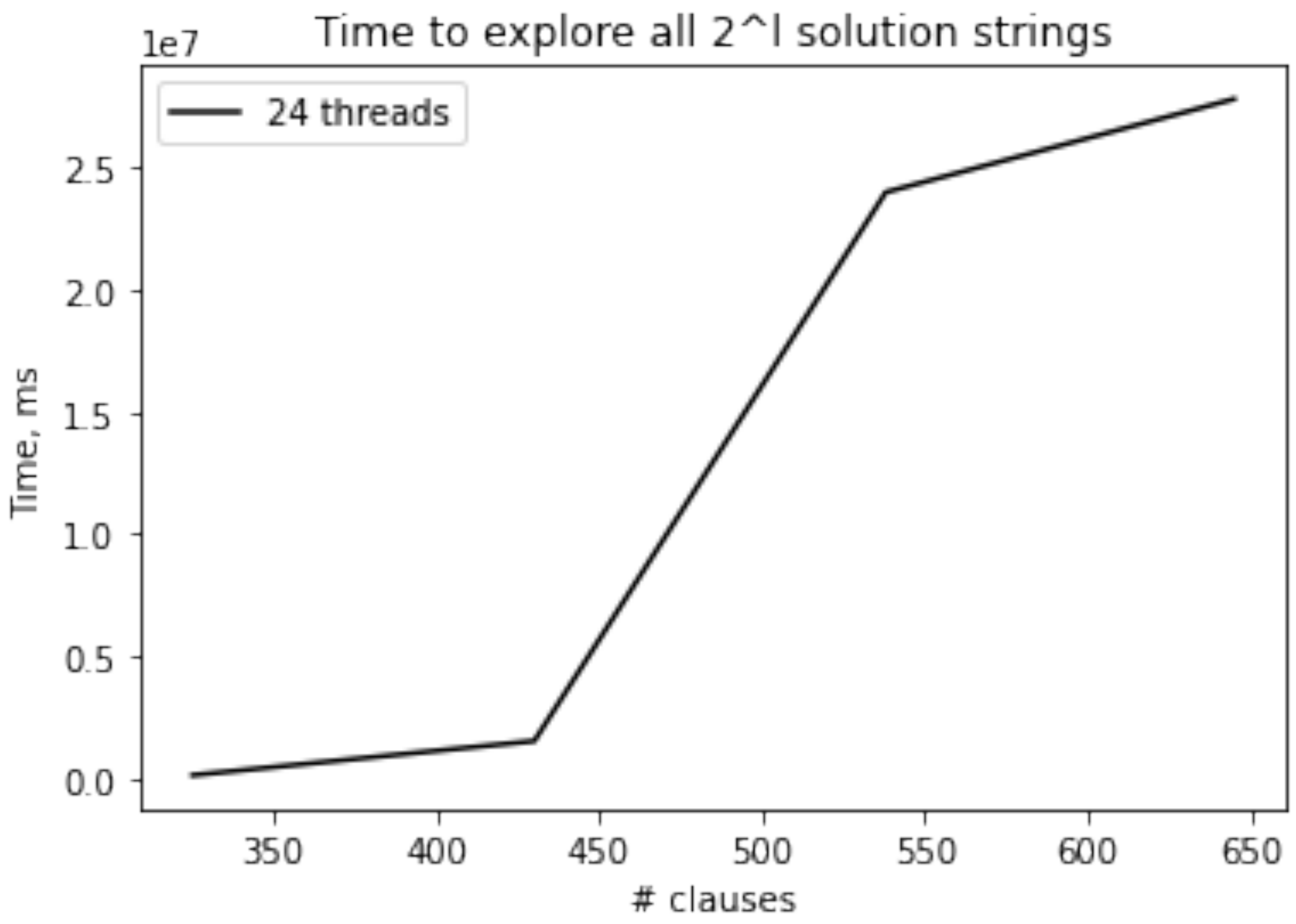


Results, Varied Problem Size, SAT



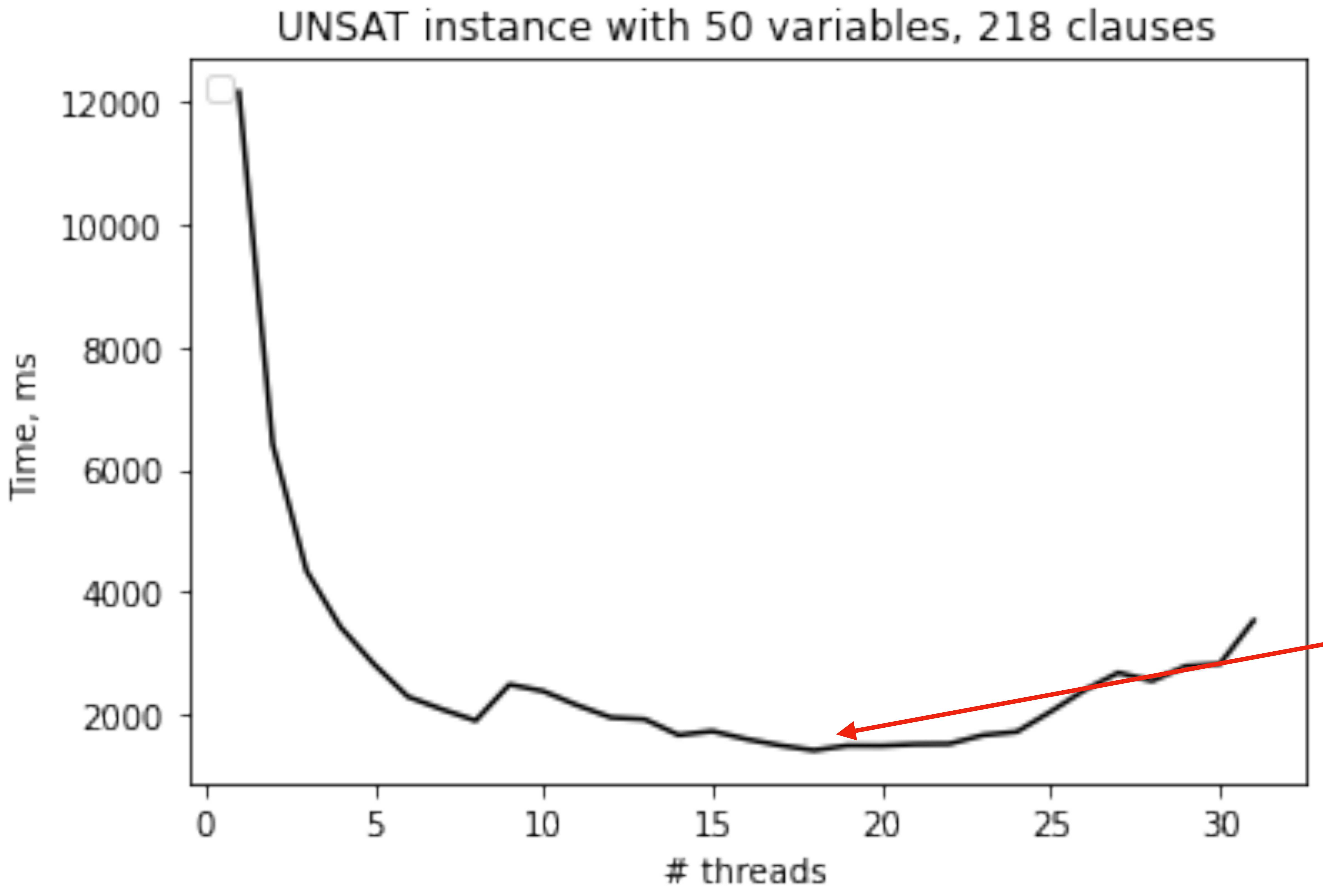


Results, Varied Problem Size, UNSAT



- Since this is a **complete algorithm**, entire solution space is checked and thus an answer of FALSE would require checking all solutions, which is computationally expensive
- mc18 was tasked with exploring 2^{150} possible solutions
- In 24 hours, serial algorithm **could not finish**
- But parallel algorithm **found the solution in 7.7 hours** using 24 threads, and then moved on to smaller problem sizes

Results, Varied Threads Count



Tests on mc20, 32 core CPU

$T_s = O(cl)$

$T_p = \Theta(\frac{c}{p}l + pl)$, row decomposition

$T_o = \Theta(p^2 l)$

For **isoefficiency**, $c \approx p^2$

$c = 218, \therefore p_{\text{max}} \approx \sqrt{218} \approx 15$

After approximately 15 threads, T_p starts increasing as shown in the experimentation.

Therefore, since number of clauses was not increased, the performance decreases as number of threads increase.

Design and Implementation Decisions Made

- My initial objective was to create a SIMT algorithm for unit propagation, and thus off-load this algorithm to GPU. Therefore, representing the input as a matrix and tread the Unit Propagation algorithm as a dense matrix algorithm would make it intuitive to implement on SIMT clusters.
- With the min-max search algorithm, one can also enable an additional optimization called **"Pure Literal Elimination"** which is powerful optimization axiom similar to unit propagation, but **searches for +1 and -1 across rows instead of columns**.
- It is assumed that number of literals << number of clauses, but when the number of literals explodes due to, say, reduction from a complex NP-hard problem, then one can re-enable column stripping. Therefore the algorithm is designed for the worst cases, but implementation allows for flexibility in terms of stripping.

Conclusion

- Presented decomposition techniques for finding unit clauses in a CNF formula for SAT solving
- Presented a parallel search technique to prove that a clause is not a unit clause (min-max search)
- Analyzed drawbacks of division of literals across processors
- Analyzed impact of division of clauses across processors
- Implemented a new algorithm that performs unit propagation in parallel with SAT checking
- Implemented a SAT solver that uses bit sets for data representation

Future Work

- Port the algorithm to CUDA
- Proposed improvements on 2D blocked decomposition:
    - Each processor is assigned two 64 bit integer to find min/max literals in
    - Finding lowest and highest set bits in a fixed-size bit-set can be performed in constant time
- Generate larger datasets, where number of literals > number of clauses
- Parallelize conflict checking
- Implement Pure Literal Elimination, that is, min/max search across clauses instead of across literals