PHYSICAL SHARING OF CLAUSES IN PARALLEL SAT SOLVERS

por

Juan Luis Olate Hinrichs

Patrocinante: Leo Ferres

Memoria presentada
para la obtención del título de

INGENIERO CIVIL INFORMÁTICO

Departamento de Ingenería Informática y Ciencias de la Computación

de la

UNIVERSIDAD DE CONCEPCIÓN

Concepción, Chile
Octubre, 2011

Abstract

Your abstract goes here...

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the most well-known problems in computer [4] science is the satisfiability (SAT) problem. This is because this was the first problem to be proved to be NP-complete [2], proof known as the Cook-Levin theorem[1]. One year later, in 1972, Karp proved in [6] that many common combinatorial problems could be reduced in polynomial time to instances of the SAT problem, thus drawing even more attention to SAT problems by the scientific community. Because many combinatorial problems can be reduced to SAT, it is not strange to find many practical problems with useful applications (such as circuit design and automatic theorem proving) that could be solved if there was an efficient algorithm to solve the SAT problem. Unfortunately, because of the NP-complete nature of SAT, such algorithm has not been found yet, but also has not been proven to be in-existent. Many researchers suspect such efficient algorithm to solve all SAT instances does not exist, so instead of trying to solve the NP-complete problem, they try to improve the current SAT solving algorithms.Over the years, SAT solvers have shown impressive improvement, the first complete algorithm, the Davis Putnam algorithm [10], was very limited and could only handle problems with around ten variables. Today, modern SAT solvers can handle instances with millions of variables, making such solvers suitable even for industrial application. In the next chapter we will point out the main features that have improved SAT solvers significantly.

In the last decade parallel computing has become increasingly popular. As CPU manufacturers have found difficult and expensive to keep increasing the clock speed of processors, they have instead turn to increase the number of cores each chip has. Unfortunately, if the algorithms are not thought to be run in parallel, more cores will bring small improvements. This is the reason why there is a growing concern to

[1]They both proved it independently.

parallelize algorithms so that they can take advantage of many-cores architectures of today's computers. In SAT solving it is no different. The annual SAT competition [1], an event to determine which is the fastest SAT solver, has two main categories; sequential SAT solvers and parallel SAT solvers. In the last years parallel SAT solvers have outperformed sequential solvers in total wall clock time, so the interest in parallel solvers has grown, new designs and approaches have been explored for this kind of solvers. One of the most successful approaches to implement a parallel SAT solver is the portfolio approach. This approach is basically to run different solvers in parallel and wait for one of them to solve the problem. It's a very simple and straight forward approach of parallelization, but we have also encountered one drawback to it: as we add more solvers to different cores of a single chip, the overall performance of the parallel solver decreases in around 20-40%. In this work we will attempt to find the source of this problem and explore possible solutions to it.

[1] www.satcompetition.org

Chapter 2

Background and Related Work

2.1 SAT solvers

2.1.1 The SAT problem

Given a set of boolean variables Σ, a literal L is either a variable or the negation of a variable in Σ, and a *clause* is a disjunction of literals over distinct variables[1]. A propositional sentence is in *conjunctive normal form* (*CNF*) if it has the form $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$, where each α_i is a clause. The notation of sentences in CNF we will be using are sets. A clause $l_1 \vee l_2 \vee ... \vee l_m$, where l_i is a literal, can be expressed as the set $\{l_1, l_2, ..., l_m\}$. Furthermore, the CNF $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$ can be expressed as the set of clauses $\{\alpha_1, \alpha_2, ..., \alpha_n\}$. With these conventions, a CNF Δ is valid if Δ is the empty set: $\Delta = \emptyset$. A CNF Δ will be inconsistent if it contains the empty set: $\emptyset \in \Delta$. Given a CNF Δ, the SAT problem is answering the question: Is there an assignment of values for variables in Σ, such that Δ evaluates to true? The NP-completeness of this question lies in the combinatorial nature of the problem; to solve it one would need to try all different assignments of variables in Σ, the number of possible assignments grows exponentially as $|\Sigma|$ grows.

2.1.2 Resolution

The resolution inference rule [11] is defined as follows. Let P be a Boolean variable, and suppose that Δ is a CNF which contains clauses C_i and C_j, where $P \in C_i$ and $\neg P \in C_j$. The resolution inference rule allows us to derive the clause $(C_i - \{P\}) \cup (C_j - \{\neg P\})$, which is called a *resolvent* that is obtained by *resolving* C_i and C_j. A simple example is the CNF $\{\{A, \neg B\}, \{B, C\}\}$. Applying resolution to this two clauses would derive the clause $\{A, C\}$, which would be called a B-*resolvent*.

[1]That all literals in a clause have to be over distinct variables is not standard.

Resolution is incomplete in the sense that it is not guaranteed to derive every clause that is implied by the CNF, but it is *refutation complete* on CNFs. It is guaranteed that resolution will derive the empty clause if the given CNF is unsatisfiable.

Unit resolution is an important special case of resolution. It's a resolution strategy which requires that at least one of the resolved clauses has only one literal. Such clause is called a *unit clause*. The importance of unit resolution does not rely on its completeness (it's actually not refutation complete, as resolution is), but one can apply all possible unit resolution steps in time linear to the size of a given CNF. Its efficiency makes it a key technique employed by modern solvers.

2.1.3 Conditioning

Conditioning a CNF Δ on a literal L consists of replacing every occurrence of L by the constant **true**, replacing $\neg L$ with **false**, and simplifying accordingly. The result of conditioning Δ on L will be denoted by $\Delta|L$ and can be defined as follows:

$$\Delta|L = \{\alpha - \{\neg L\} | \alpha \in \Delta, L \notin \alpha\}$$

This means that the new set of clauses $\Delta|L$ will be all the clauses in Δ that do not contain L, but with the literal $\neg L$ removed. The clauses that contain L are removed because they are now satisfied, since we made L **true**. $\neg L$ is removed from the remaining clauses because it was set to **false** and no longer has any effect.

The definition of conditioning can be extended to multiple literals. For example, the CNF:

$$\Delta = \{\{A, B, \neg C\}, \{\neg A, D\}, \{B, C, D\}\}$$

can be conditioned as $\Delta|CA\neg D = \{\emptyset\}$ (an inconsistent CNF). Moreover, $\Delta|\neg CD = \emptyset$ (a valid CNF).

2.1.4 Satisfiability by search: The DPLL algorithm.

2.1.4.1 Search trees

One way to picture the search for a possible assignment of variables that satisfies the CNF formula is to use a tree. For example, given $\Sigma = \{A, B, C\}$ and $\Delta = \{\{\neg A, B\}, \{\neg B, C\}\}$, figure 2.2 shows the search tree for this CNF. Each node of the tree represents a variable, for each level we have a different variable. The branches are the different truth values the variable can be assigned. Each w_i represents a possible truth assignment of the variables in Σ. Note that w_1, w_5, w_7 and w_8 are all assignments that satisfy the CNF, while w_2, w_3, w_4 and w_6 do not.

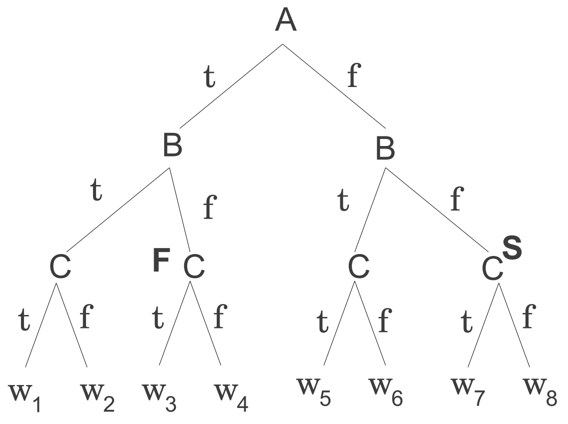

Figure 2.1: A search tree for the CNF $\Delta = \{\{\neg A, B\}, \{\neg B, C\}\}$.

2.1.4.2 Depth-search-first algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) [3] algorithm is the base of all modern SAT solvers. Many refinements have been made to this algorithm over the last decade, which have been significant enough to change the behaviour of the algorithm, but it is still important to know it for understanding modern solvers. Figure 2.2 shows a search tree with three variables. We can observe that the leaves of this tree and in one-to-one correspondence with all the possible true assignments of variables involved, so testing satisfiability can be viewed as searching for a leaf node that satisfies the CNF. Another important observation is that the depth of the tree is n, where n is the number of boolean variables, so performing a depth-first-search would be best

Algorithm 1: Depth-first-search algorithm

```
1  DEPTH-FIRST-SEARCH(CNF $\Delta$,depth $d$):
2  if $\Delta = \{\}$ then
3      return $\{\}$.
4  else if $\{\} \in \Delta$ then
5      return UNSATISFIABLE
6  else if L=DEPTH-FIRST-SEARCH($\Delta|P_{d+1}, d+1$) $\neq$UNSATISFIABLE then
7      return $\mathbf{L} \cup \{P_{d+1}\}$
8  else if L=DEPTH-FIRST-SEARCH($\neg\Delta|P_{d+1}, d+1$) $\neq$UNSATISFIABLE then
9      return $\mathbf{L} \cup \{\neg P_{d+1}\}$
10 else
11     return UNSATISFIABLE
```

to explore the tree. Algorithm 1 performs a depth-first-search, which is the base of the DPLL algorithm, using the conditioning operator on CNFs to remove clauses or reduce their size.

Consider the CNF:

$$\Delta = \{\{\neg A, B\}, \{\neg B, C\}\},$$

and the search node labelled with **F** in Figure 2.2. At this node, Algorithm 1 will condition Δ on literals $A, \neg B$, leading to:

$$\Delta|A, \neg B = \{\{\mathbf{false}, \mathbf{false}\}, \{\mathbf{true}, C\}\} = \{\{\}\}.$$

The algorithm will detect that at this internal node that there is a contradiction, hence concluding that neither $w3$ or $w4$ are models of Δ, without having to visit them explicitly. The algorithm can also detect success at internal nodes, implying that all assignments from that particular node are models of the CNF.

2.1.4.3 Unit Resolution

Consider the tree of Figure 2.2 and the CNF:

$$\Delta = \{\{\neg A, B\}, \{\neg B, \neg C\}, \{C, \neg D\}\}.$$

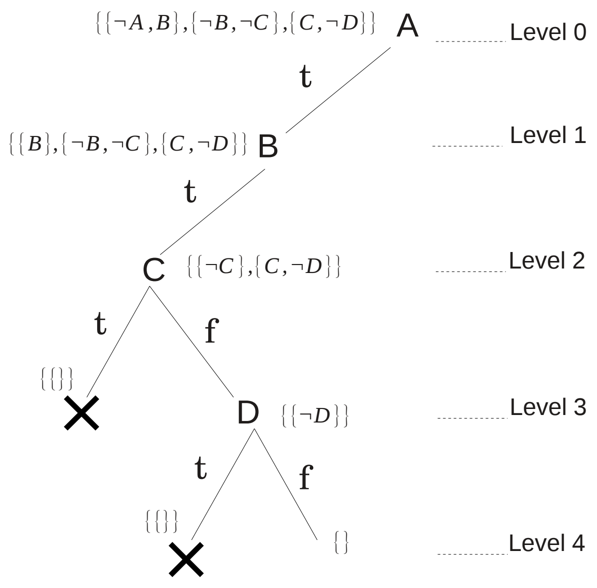

Figure 2.2: A termination tree, where each node is labelled by the corresponding CNF. The last node visited during the search is labelled with {}. The label X indicates the detection of a contradiction at that node.

Consider also the node at Level 1, which results from setting variable A to **true**, and its corresponding CNF:

$$\Delta|A = \{\{B\}, \{\neg B, \neg C\}, \{C, \neg D\}\}.$$

Algorithm 1 cannot declare early success or early failure, because the CNF is neither empty, nor contains the empty clause, reason why it keeps searching below Level 1 as shown in Figure 2.2. However, we will now show that by using unit resolution, we can declare success success and end the search at Level 1.

The *unit resolution technique* (also called *unit propagation*) is very simple: Before we check tests for success or contradictions, we first collect all unit clauses. Then we assume that the variables which make these clauses unit are set to satisfy the unit clauses. Finally we simplify the CNF and check for success or failure.

To incorporate unit resolution into Algorithm 1, we will introduce a function UNIT-RESOLUTION, which applies to a CNF Δ and returns two results:

- **I**: a set of literals that are either present as unit clauses in Δ, or were derived from Δ by unit resolution.

- Γ: a new CNF which results from conditioning Δ on literals $\mathbf{I}$.

For example, if the CNF

$$\Delta = \{\{\neg A, \neg B\}, \{B, C\}, \{\neg C, D\}, \{A\}\},$$

then $\mathbf{I} = \{A, \neg B, C, D\}$ and $\Gamma = \{\}$. Moreover, if

$$\Delta = \{\{\neg A, \neg B\}, \{B, C\}, \{\neg C, D\}, \{C\}\},$$

then $\mathbf{I} = \{C, D\}$ and $\Gamma = \{\{\neg A, \neg B\}\}$.

2.1.4.4 DPLL algorithm

Algorithm 2: DPLL(CNF Δ): returns a set of literals or UNSATISFIABLE

```
1  $(\mathbf{I}, \Gamma)$ = UNIT-RESOLUTION($\Delta$)
2  if $\Gamma = \{\}$ then
3      return I.
4  else if $\{\} \in \Gamma$ then
5      return UNSATISFIABLE
6  else
7      choose a literal $L$ in $\Gamma$
8      if L = DPLL($\Gamma|L$) $\neq$ UNSATISFIABLE then
9          return $\mathbf{L} \cup \mathbf{I} \cup \{L\}$
10     else if L = DPLL($\Gamma|\neg L$) $\neq$ UNSATISFIABLE then
11         return $\mathbf{L} \cup \mathbf{I} \cup \{\neg L\}$
12     else
13         return UNSATISFIABLE
```

The DPLL algorithm (Algorithm 2) is a refinement of the depth-first-search algorithm. The first change made is that it adds unit resolution in line 1. Also, we no longer assume that variables are examined in the same order as we go down the search tree and we no longer assume that a variable is set to **true** and then to **false**. The choice of a literal L on line 7 can have a dramatic impact on the running time of DPLL. This is where inference comes into play when solving a SAT instance with a DPLL based algorithm, heuristics and random factors are commonly used at this point.

2.1.4.5 Chronological backtracking

When we detect a conflict at level l of the search tree, we have to rewind back to level $l-1$, undoing all assignments done in the process. Then we try another value at level $l-1$, if none remains to be tried, we then go back further to level $l-2$, and so on. If we ever reach level 0 and each value there leads to a contradiction, we will know that the CNF is inconsistent. This type of backtracking is called *chronological backtracking*, because the we move back in the same order we got there; one level at a time.

2.1.4.6 Non-Chronological backtracking

The problem with chronological backtracking, which is the one used in the DPLL algorithm, is that contradictions (when an assignment we have done does not satisfy the CNF) that trigger the backtrack often have valuable information. Such information could lead us to learn new clauses and backtrack more levels than just one, thus saving time in the search. For example, let's say that we have a CNF Δ with variables A, B, C, D. Let's us also say that, because of the clauses in Δ, the CNF will always be inconsistent when A is set to **true**. Unfortunately, the DPLL algorithm might not realise this beforehand, because it uses unit resolution which is not refutation complete. So the algorithm might only conclude that A set to **true** will not satisfy Δ after assigning all possible values to B, C, D. If we could detect that A is not part of any solution, we could avoid going any further in that branch of the search tree and learn that information as a new clause (in our example we would learn the clause $\{\neg A\}$ to enforce A to be **false**). Learning such clauses is called *conflict-driven clause learning* (CDCL) [16, 8].

We will not discuss in detail how such new clauses are decided or how can we decide which level to backtrack to after a contradiction is found. It suffices to know that after a contradiction is detected, we can decide a new clause to be learnt and which level to backtrack to. Also note that adding CDCL features to the DPLL algorithm will not affect its soundness or completeness.

2.1.5 Modern CDCL solvers

Most modern solvers today are CDCL SAT solvers. Given a CNF φ, a partial assignment of variables ν, Algorithm 3 outlines the general structure of a CDCL SAT solver, where x is a variable, v a truth value and β a decision level. We will shortly explain the main functions of this algorithm.

Algorithm 3: Typical CDCL algorithm

```
   Input: A CNF $\varphi$ and a variable assignment $\nu$
1  if (UnitPropagation($\varphi$,$\nu$)==CONFLICT) then
2      return UNSAT.
3  $dl \leftarrow 0$
4  while (not AllVariablesAssigned($\varphi$,$\nu$)) do
5      $(x,v)$=PickBranchingVariable($\varphi$,$\nu$)
6      $dl \leftarrow dl + 1$
7      $\nu \leftarrow \nu \cup \{(x,v)\}$
8      if (UnitPropagation($\varphi$,$\nu$)==CONFLICT) then
9          $\beta$=ConflictAnalysis($\varphi$,$\nu$)
10         if ($\beta < 0$) then
11             return UNSAT
12         else
13             Backtrack($\varphi$,$\nu$,$\beta$)
14             $dl \leftarrow \beta$
15 return SAT
```

- UnitPropagation consists of iteratively deducting the truth value of variables. The values are deduced by logical reasoning on φ and ν. We already discussed this function in the previous sections.

- PickBranchingVariable consists of selecting a variable to assign, and the respective value. Heavily relies in heuristics/random factors to pick variables.

- ConflictAnalysis consists of analyzing the most recent contradiction and learning a new clause from it. It returns the decision level to backtrack to (non-chronological backtracking).

- Backtrack undoes variable assignments and backtracks to a previous decision level as computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned a truth value.

2.1.5.1 The two watched literals lazy data structure

Data structures play a fundamental role in the performance of CDCL SAT solvers. Many improvements have been achieved over the last years, but one of the most noticeable ones is the implementation of the so called *lazy data structures* (lazy because they are not accurate). There are different types of lazy data structures used in modern SAT solvers, they mainly try to address cache performance problems when the solver is performing unit resolution (which adds up to about 70% of the execution time of a SAT solver [**?**]). The two main approaches to implement lazy data structures are the head/tail lists used in Sato [15] and the two watched literals used in Chaff [9]. We will explain the later in detail, as it's widely implemented in most modern CDCL SAT solvers.

Unit propagation is a key function to the speed of a SAT solver. We need fast ways of identifying which clauses become unit when assigning variables. For example, if we have the clause

$$\{\neg A, B, C, \neg D, E\}$$

and the variable assignment

$$A = \textbf{true}, B = \textbf{false}, C = \textbf{false}, E = \textbf{false},$$

we would need to identify somehow that this clause is unit on variable D. The two watched literal data structure addresses this problem and makes backtracking very efficient. Instead of trying to keep precise information on the complete state of the clause, the two watched literal strategy only keeps track of two literals in each clause. Such two literals are the watched literals of a clause. Let's take as an example the previous clause, as shown on Figure 2.3. This time we will assume that no variable assignments have been done yet and that the two watched literals are $\neg A$ and B.

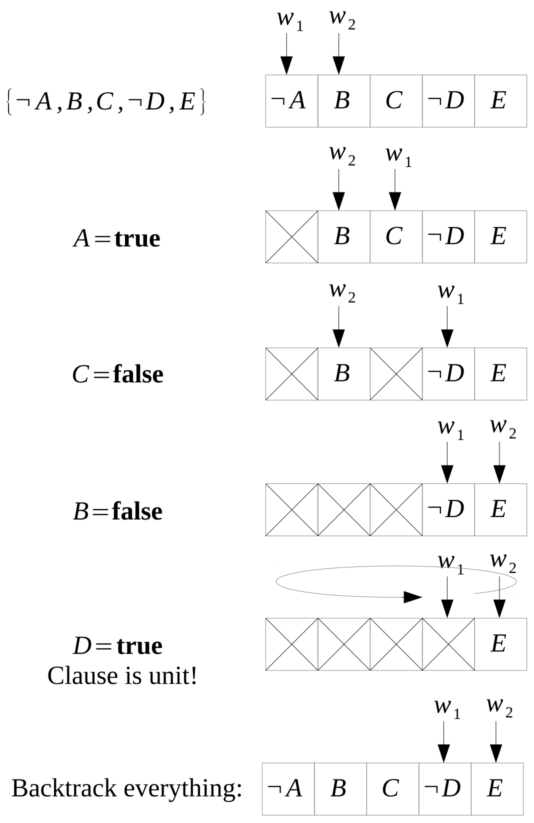

Figure 2.3: Operation of the two watched literal data structure.

Suppose that on the next decision level variable A is assigned as **true**. We would check the watched literals and realize that this clause might be unit, because now there is only one watched literal which is unassigned and the other doesn't satisfy the clause. Remember that we are only aware of the watched literals and not the rest of them. To check if the clause is really unit, we will attempt to change the first watched literal to an unassigned variable in the clause. We could search to the right and find that literal B is already being watched (the two watched literals must be different), so we keep searching to the right and find that literal C is unassigned, concluding that the clause is not yet unit. Now our two watched literals for this clause are B and C. Let's suppose now that propagation gives the value **false** to variable C, then we would again wonder if the clause is unit or not. Searching to the right for another unassigned literal to watch will find literal $\neg D$, so now our two watched literals will be B and $\neg D$. If the next assigned variable is $B =$ **false**, then one watched literal would have to search the clause and find that literal E as available. Finally, if variable D is assigned to **true**, then we would search the clause for an available literal, but won't find any (after checking all literals). Only now we can declare that this clause is unit, because there is only one unassigned watched literal and the other can't find any unassigned position. Propagation would assign **true** to variable E, because it's the only way to satisfy that clause. If a conflict is found, while propagating across the different variables, we have to undo all variable assignments after the backtracking point, but the key advantage of the two watched literal structure is that no backtracking has to be done to any watched literal reference of any clause. The clear drawback to this advantage is that, as we saw in the previous example, we have to check all literals in the clause before declaring it unit, since we only keep track of two literals at a time.

2.1.6 Parallel SAT solvers

As mentioned before, some parallel SAT solvers have performed at the top of the last SAT competitions, but even though they all fall into the parallel solvers category, their parallel strategies and implementations vastly differ from each other. We mainly classify parallel SAT solvers into two categories: Portfolio approach solvers and divide-and-conquer solvers.

The main idea behind portfolio approach solvers is the fact that different kinds of sequential solvers will perform differently for different kinds of SAT problems. The portfolio approach is a very straight forward strategy: They run a group of sequential solvers in parallel, each with different heuristic random values and/or different search strategies. The time they take to solve the problem will be the time of the fastest solver in the group of solvers running in parallel. Although all portfolio approach solvers share this same principle, they also have quite different kinds of implementations. We identify in this group the solvers that are pure portfolio approach, the ones that share clauses only logically, and the ones that share clauses physically and logically.

Solvers which are pure portfolio approach have the most simple design. They run completely independent solvers in parallel and wait for one of them to give an answer. Despite their simplicity, the solver ppfolio [12], a pure portfolio approach solver, was the winner of the crafted and random categories, and second place in the application category of the 2011 SAT competition of parallel solvers.

On the other hand, we have more elaborated portfolio approach solvers, which can also share clauses logically between their different solvers. One of the advantages of CDCL solvers is the fact that they can learn new lemmas as they solve a SAT problem. These new lemmas will provide additional information during the solution search, so that the solver doesn't fall into previous fruitless search paths (there are also some drawbacks to adding new lemmas, which are addressed by clause database cleanups). The idea is that different solvers running in parallel can share their learned lemmas so that they all benefit from what other solvers have learned and improve their own search. An example of these kind of solvers is ManySAT [5], which won the 2009 SAT competition in the parallel solver application category. ManySAT has its own sequential state-of-the-art SAT solver and runs different instances of it in parallel, using different VSIDS [14] heuristics (branching heuristics) and restart policies for each of it, both of which account for random factors in the solver. The difference with pure portfolio approach solvers, is that ManySAT also shares learned lemmas between solving threads. It is called logical sharing of clauses, because the lemmas are passed as messages between threads and they never share the same physical information in memory. The advantage of logical sharing is that it is easier to implement message passing between threads, than having threads reading and modifying the same memory locations, which often requires locks that could hinder the overall solver performance. One of the best parallel performing solvers, Plingeling [1], also shares clauses logically. It is a very weak sharing though, since it only shares unit lemmas and it does so through message passing, using a master thread to coordinate messages between worker threads.

Portfolio approach solvers that share clauses physically have the same strategy as mentioned before, but they share clauses by allowing threads to access the same memory locations, instead of message passing. One solver in this category is SArTagnan [7], which shares clauses logically and physically.

Divide-and-conquer solvers do not try to run different solvers in parallel, they run one solving instance, but try to parallelize the search and divide it between the different threads. A common strategy to divide the search space is to use guiding paths. A guiding path is a partial assignment of variables, which restricts the search space of the SAT problem. A solver that divides its search space with guiding paths will assign threads to solve the CNF with the given partial assignment from the guiding path the thread was assigned with. Once a thread finishes searching a guiding path with no success, it can request another to keep searching. MiraXT [13] is a divide-and-conquer SAT solver which uses guiding paths. Moreover, different threads solving different guiding paths also share a common clause database, in which they store their learned lemmas. This is another example of physical clause sharing.

The work will mainly consist of building a parallel portfolio approach solver, which uses a similar shared clause database system as proposed by MiraXT. The main difference of our work with MiraXT is that we are interested in portfolio approach solving, so we will use the clause database model of MiraXT, but not its divide-and-conquer guiding path strategy.

2.1.6.1 MiraXT

2.2 Modern computers memory hierarchy

Computers today usually have three levels of memory cache and the main memory. The processor can have multiple cores in it and cores can run multiple threads in them. The difference between a core and a thread is that cores have separate copies of almost all the hardware resources. The cores can run independently unless they are using the same resource (for example the connection to the outside) at the same time. Threads, on the other hand, share almost all of the processor resources. When a thread, which is running on a core, needs to fetch data, it first tries to look for it in the first level cache (the L1 cache[1]), if the data is not there, then it tries to find it in the level 2 cache (L2 cache). If the data is still not there, it then tries to fetch it from the level 3 cache (L3 cache) and if that fails too, it goes up to main memory to get the data. If it still isn't in main memory, then it has to retrieve it from the hard disk. We should notice that this hierarchy involves increasing fetch times as we

[1]There is L1 data cache and also L1 instruction cache, we will be referring to L1 data cache

go up. Getting data from the L1 cache is much faster than getting it from L2, and getting data from L2 is much faster than getting it from main memory. The problem with lower level memory is that, because of the technology and costs involved, they are much smaller. So the big picture is that at lower levels we have faster and smaller memories, and at higher levels we have massive and slower memory storages. Figure 2.5 is a schematic of today's computer memory hierarchy. To get an idea of the times involved in accessing data from different memory storages, we present the following table of costs associated with hits and misses, for an Intel Pentium M:

To where	Cycles
Register	≤ 1
L1	3
L2	14
Main Memory	240

Another important topic to discuss is how these memory caches operate. When data is requested and it's not found in any of the three caches, it has to be loaded from main memory. Data is not transferred individually, instead, a fixed amount of bytes containing the data (or part of it if it's large) is fetched. This fixed amount of bytes is called a *word* or *cache line*. Intel uses an *inclusive* memory cache protocol, which loads a requested word from main memory into all cache levels. Because transferring data from main memory is so costly (compared to any cache level), we would like to transfer the highest amount of useful data from main memory to cache every time. Since the access times to any cache are so little compared to main memory, we will assume the bottleneck of data-fetching performance is main memory.

Data-fetching performance is not only an issue that concerns hardware designers, but also programmers. Depending on how the program is written, the same task (input-output wise) might take much longer if one is not aware about how memory behaves. Since we want the highest amount of useful data in each word transferred, the data structures in which the programmer decides to store data will have a dramatic impact in cache performance. Consider, for example, the following problem: Write a program that sums all elements in a matrix size $m \times n$ of integers. Algorithm 4 and 5 both solve the problem, and apparently with the same level of efficiency, from an

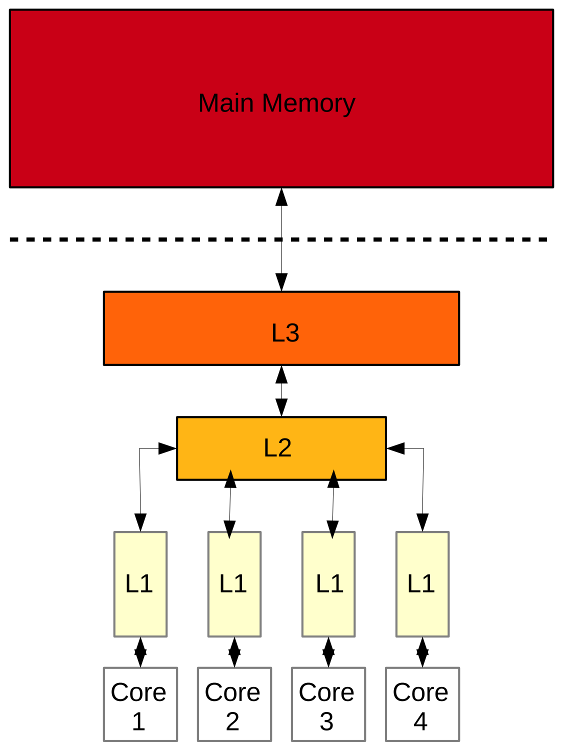

Figure 2.4: Typical memory hierarchy of modern computers.

algorithmic point of view at least. However, the actual results of both algorithms in practice have a vast difference in total solving time. The following table shows the results for both algorithms implemented in C, for a matrix of size 17000 × 17000:

Algorithm	Elapsed time (seconds)
Row sum	2.5
Column sum	37.5

Algorithm 4: Row sum of elements

Input: A matrix M of size $m \times n$, which elements are integers

```
1 $sum \leftarrow 0$
2 for $i \leftarrow 0$ to $m$ do
3     for $j \leftarrow 0$ to $n$ do
4         $sum = sum + M[i][j]$
5 return $sum$
```

Algorithm 5: Column sum of elements

Input: A matrix M of size $m \times n$, which elements are integers

```
1 $sum \leftarrow 0$
2 for $i \leftarrow 0$ to $n$ do
3     for $j \leftarrow 0$ to $m$ do
4         $sum = sum + M[j][i]$
5 return $sum$
```

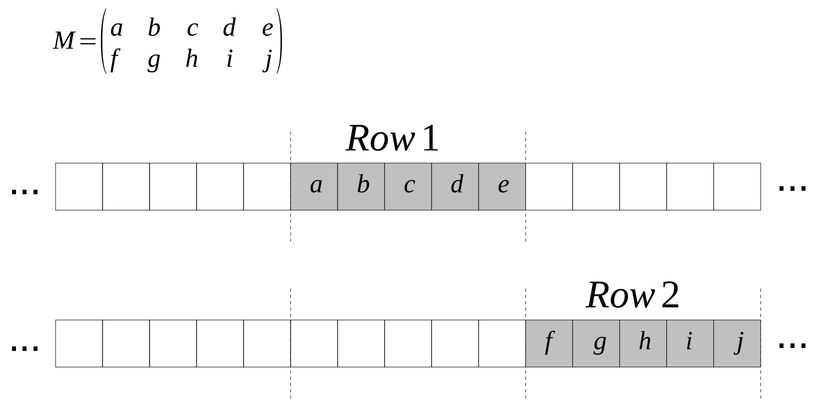
$M = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \end{pmatrix}$
Row 1
a b c d e
Row 2
f g h i j
...

Figure 2.5: Typical memory hierarchy of modern computers.

Chapter 3

Important stuff...

3.1 Cache performance in simple parallel-shared-memory programs

The main motivation behind this work is our experimental evidence that when threads in different cores share memory, the cache performance improves compared to having threads with its own memory. Our first experiment will be a program that performs simple arithmetic operations on a big matrix. We will have two instances for this experiment. The first will be different threads performing the same arithmetic operations over the same matrix (hence yielding the same result), but every thread will hold its own copy of the big matrix. The second instance will be the same experiment, but this time the different threads will perform the arithmetic operations over the same matrix (regarding physical memory location). For this experience, we will only use read-only operations over the matrix, to keep concurrency problems out of the equation.

To measure cache performance we will use the linux tool *perf*. Table **??** shows...

3.2 Cache performance of parallel SAT solvers

3.2.1 Barcelogic

3.2.2 Plingeling

3.3 A parallel portfolio approach SAT solver with physicaly shared clause database: AzuDICI

3.3.1 General outline

3.3.2 Data structures involved

3.3.3 Propagation

3.3.4 Experimental results

Chapter 4

Conclusions

Bibliography

[1] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1%2B2%2B3%2B6.pdf`, 2010. [Online; accessed 01-April-2012].

[2] S. A. Cook. The complexity of theorem-proving procedures. *proc. 3rd ann. ACM symp. on theory of computing*, pages 151–158, 1971.

[3] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[4] Ulrich Drepper. What every programmer should know about memory, 2007.

[5] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.

[6] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[7] Stephan Kottler and Michael Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[8] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153.

[9] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[10] Davis Putnam and Martin Hillary. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[11] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[12] O. Rousell. Description of ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`, 2010. [Online; accessed 01-April-2012].

[13] Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: parallel sat solving with threads and message passing. *Journal of Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.

[14] Jaeheon Yi. The effect of vsids on sat solver performance. 2009.

[15] Hantao Zhang. Sato: An efficient propositional prover. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 1997.

[16] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.