15-418 F22: Parallel Computer Architecture and Programming
Term Project: CPU and GPU Accelerated Parallel SAT Solver

Name: Jiarui Wang, Tejas Badgujar
Andrew ID: jiaruiwa, tbadguja

Project Github URL

1 Summary

We implemented two parallel SAT solvers that are based on divide and conquer and portfolio method. The divide and conquer algorithm explored how different workload assignment will impact the workload balance and overall speedup. We also explored CUDA divide and conquer algorithm that utilize the high throughput hardware architecture. Our CPU divide and conquer algorithm achieved 7.09x speedup while our GPU implementation achieved over 1000x speedup on the GHC machine. In portfolio method, we implemented DPLL and CDCL algorithm, two standard serial SAT solver and combined them together in the portfolio method to achieve much better performance.

2 Background

The algorithm will take in SAT problems in 3CNF form. A conjunctive normal form (CNF) is where a boolean formula is only consist of conjunctive of clauses. A clause is a disjunction of literals. Literals are either a variable or the negation of a variable.

To put it more concretely, x is a variable that could be assigned true or false value. Therefore, the following are both literals:

$$x, \neg x$$

Similarly, x, $\neg x$ and y are literals and if we connect them together with $\vee$ to form a disjunction, we have a clause:

$$x \vee \neg x \vee y$$

To put everything together, if we connect the clauses together with $\wedge$ to form a conjunction of disjunctions, we have a boolean formula that is in conjunctive normal form (3CNF is just a special case of CNF where each disjunctive clause only contains three literals):

$$(x \vee \neg x \vee y) \wedge (z \vee \neg y \vee x) \wedge (\neg x \vee \neg z \vee \neg y)$$

Note that the CNF above is satisfiable with the assignment $x = true, y = false, z = false$ since if we plug

in the assignment into the CNF, we have:

$$\begin{aligned}
&(x \vee \neg x \vee y) \wedge (z \vee \neg y \vee x) \wedge (\neg x \vee \neg z \vee \neg y) \\
\Longleftrightarrow &(T \vee \neg T \vee F) \wedge (F \vee \neg F \vee T) \wedge (\neg T \vee \neg F \vee \neg F) \\
\Longleftrightarrow &(T \vee F \vee F) \wedge (F \vee T \vee T) \wedge (F \vee T \vee T) \\
\Longleftrightarrow &T \wedge T \wedge T \\
\Longleftrightarrow &T
\end{aligned}$$

We use an integer array to represent SAT problems in 3CNF form. Every three integer in the array forms a CNF clause and a negative sign represent the boolean negation operator. For example, a CNF clause containing 3 variables will be represented in the following manner (assuming x maps to 1, y maps to 2 and z maps to 3)

$$\begin{aligned}
(x \vee z \vee \neg y) \wedge (z \vee x \vee y) \wedge (y \vee x \vee \neg z) &\Longrightarrow (1 \vee 3 \vee \neg 2) \wedge (3 \vee 1 \vee 2) \wedge (2 \vee 1 \vee \neg 3) \\
&\Longrightarrow (1 \vee 3 \vee -2) \wedge (3 \vee 1 \vee 2) \wedge (2 \vee 1 \vee -3) \\
&\Longrightarrow 1, 3, -2, 3, 1, 2, 2, 1, -3
\end{aligned}$$

The example 3CNF problem can be solved by assigning true to x, false to y and false to z. This corresponds to the bit pattern of $0b001$ which is just integer 1.

The algorithm will output the satisfiability of the SAT problem. For some implementations, the program will also output the boolean assignment, but most of the algorithm implemented in the project will only determine the satisfiability of the problem.

2.1 Divide and Conquer

In divide and conquer method, due to the natural limit of brute force algorithm, we have decided that in this portion of the project, we probably do not need data structures that support an unlimited amount of variables. Therefore, we have decided to use a 64-bit unsigned integer's bit pattern to represent the boolean assignment of variables.

Since variables in our SAT problem are all represented by integers, we could use bitwise operator to extract out the bit that represents their boolean assignment. For example, variable 5's boolean value will be represented by the 4-th bit in the integer (there is no variable 0 so we will start at 1). To get the next assignment, we could just increment the integer by 1. For x number of variables, it would have 2^x possible assignments and all of that will be covered in the numerical range from 0 to $2^x - 1$.

While evaluating one assignment takes almost no time, the search space is exponentially large. For x number of variables, there will be $2^x - 1$ possible boolean assignments to them. Covering this entire search space and evaluating every possible assignment in it is very computationally expensive and could benefit greatly from parallelization.

One great thing about the SAT problem search space is that there is zero dependency between evaluation. Therefore, this problem is perfect for parallelism. It is not data-parallel since every single worker will have a copy of the integer array that represents the SAT problem. The program should have a lot of time and space locality since at every evaluation, the worker will access the same integer array, meaning that a worker will access the same data over and over again during its search.

The algorithm is also friendly to SIMD since at every iteration of the loop, the algorithm will do the same bitwise operation and access the same data. Nevertheless, since different boolean assignment will take different amount of time to evaluate, some loop iteration may terminate earlier than others. Some may fail on the first clause while others may fail on the last clause and this will be problematic to SIMD since it will cause some vector lanes to idle while other vector lanes are still working.

2.2 DPLL, CDCL and Portfolio Method

Moving on from the parallelization of the naive algorithm, we now start to discuss the more advanced algorithms used as the foundation for modern SAT solvers. For the scope of this project, we will specifically discuss the DPLL (Davis–Putnam–Logemann–Loveland) and CDCL (Conflict-Driven Clause Learning) approaches to boolean satisifiability, and how it may be possible to piece together SAT solving methods through the portfolio method.

2.2.1 DPLL

Compared to the naive implementation which decides on a full assignment initially and then proceeding to test whether or not the assignment satisfies the formula, both DPLL and CDCL operate using partial evaluations which eventually get filled up with assignments based on decisions or boolean analysis upon the formula-assignment state.

Specifically, in DPLL, there are two straightforward procedures that occur in terms of boolean analysis: unit clause propagation and pure literal elimination. Combining these two procedures gives us DPLL in almost its entirety: the algorithm consists of backtracking search through partial assignments while interleaving unit clause propagation and pure literal elimination after a new variable has been decided. As for the specifics to how we implement this and what unit clause propagation and pure literal elimination entail, we will discuss it more in the approach section.

2.2.2 CDCL

Building upon the ideas established in DPLL, CDCL adds in another form of analysis to help with its satisfiability search: clause learning. By analyzing the decisions made leading up to a unsatisfiable clause while searching for a satisfying assignment, the algorithm is able to add new clauses to the formula which will allow it to essentially make better, "more informed" decisions going forward. Additionally, due to the nature of the clause learning, because it combines multiple decisions done at different levels of the algorithm's search, it runs with non-chronological backtracking, as opposed to DPLL's simple chronological backtracking.

2.3 Portfolio Method

The portfolio method is a crude yet apparently decently effective way to get high utilization and some level of parallelism for SAT solving. Simply put, it involves running different algorithms or variations of an algorithm across multiple cores, with each core getting a unique variation or algorithm. When properly executed, the separate instances should end up exploring different parts of the search space. Note that this does not equate to (but could still result in) faster runtimes for the overall algorithm, as the different paths that the algorithm takes may be of roughly equal length and could end up with similar run times compared to a sequential pass-through.

3 Approach

3.1 Uniformly Random 3 CNF Generator and Solution Verifier

After some research, it seems that it is common for researchers to generate uniformly random 3SAT problems in order to benchmark their SAT solvers. Therefore, we have written a python script that will take in the number of desired variables and clauses and generate SAT problems in 3CNF form with each variable in the clause chosen uniformly from the variables. We will also pick the sign of the variable with a 50/50 chance. The test cases generated by the python scripts has proven to be valuable for us to benchmark our SAT solvers.

Following the characteristic of NP-complete problems, verifying the solution to the problem is within polynomial time and could be done easily with a python script. Therefore, we have written a verifier in python to double check the output of our 3SAT solver. On the other hand, verifying the unsatisfiability of a problem is unnecessary in python because we will essentially be implementing the same logic in python and it will have terrible performance compared to our c++ implementation.

3.2 Boolean Variable Assignment Representation in Divide and Conquer

The first and foremost challenge and design decision was how should we represent the assignment of boolean variable in our algorithm. The most straightforward approach seems to be using a vector or an array of boolean variables. But that seemed wasteful since each boolean variable will take up 1 byte which is 8 bits when we only needed 1 bit to represent the boolean assignment. At the same time, a vector or an array of boolean variables will need to be copied between threads or loop iterations to ensure that concurrent execution will not interfere with the correctness of the algorithm. Since memory operation is expensive, this will also create unnecessary overhead for our algorithm.

After some consideration we have decided to use the bit pattern of integers to represent the boolean assignment. We first designed an object in c++ that uses an array of integers to store these assignment. We have essentially implemented an big integer representation in c++ with no size limit. Nevertheless, after some more consideration we deemed this representation wasteful since the CPU divide and conquer algorithm will not be able to solve SAT problems with more than 50 variables in any reasonable amount of time. Therefore, in the final design, we have decided to just use an unsigned 64 bits integer to represent the boolean assignment of variables with each bit representing the boolean value of each variable. This is an extremely compact design since we will be able to represent the entire assignment in one single integer and to retrieve the boolean value we just need bitwise operators.

3.3 CPU Divide and Conquer

The basic serial solver will go through all possible search space until it either find a solution to the SAT problem or exhaust all possibilities. Since we represent the boolean assignment with an unsigned integer, this means the algorithm will need to evaluate assignment from 0 to $2^x - 1$ when we have x amount of variables. Since we are using the GHC machine and its CPU (i7-9700) has 8 cores with no hyper-threading, we have decided to aim for around 7 times speedup (8 times speedup is impossible due to the inherent overhead we have from parallelism).

The most obvious way to divide the search space is statically assign a chunk of the search space to each thread. Since we have 8 available cores, we will create 8 threads with each thread search through $\frac{1}{8}$ of the search space. However, this workload assignment limit us to around 6.2x speedup. We tried the interleaving pattern but while it was able to achieve better speedup in some cases, it has worse performance in other cases. By timing the execution time of each thread, we have determined that static workload assignment will create imbalance workload among threads, which will limit our speedup.

Therefore, we switched to a dynamic workload assignment implemented with a centralized work queue. The work queue is implemented with a global unsigned 64-bit integer value that represents the current assignment value. Each thread will read the global integer and increment it by the task size granularity. It will then evaluate the assignments from the value it read to the incremented value. While the centralized queue immediately gave us near perfect workload balance, we also needed to test out various task size granularity to find the right balance between workload balancing and synchronization overhead. In the end, we were able to achieve 7.09x speedup.

3.4 GPU Divide and Conquer

The GPU on the GHC machine is RTX 2080. It has 46 streaming multiprocessors each equipped with 64 INT32 cores that is capable of concurrently process 1 arithmetic operations per cycle. In total, the GPU is capable of processing 2944 arithmetic operations per cycle. Consider RTX 2080 runs at 1.710 Ghz and i7-9700 runs at 4.7 Ghz, the GPU should be able to get to around $2944 * \frac{1.71}{4.7} = 1071$ times speedup when compared to the serial implementation on CPU.

We have basically translated the same logic from a CPU thread to a CUDA thread. In our expectation, since every thread will perform the same bitwise operation and read the same data, the SIMD computing model employed by GPU should work great. The only concern is that some CUDA thread may terminate early due divergence.

Since every CUDA thread will only repeatedly read the integer vector that represents the SAT problem and the integer vector should be able to fit inside the cache. We have decided to not utilize the block shared memory and let one block contain 32 threads. In other word, one block will be one warp.

The GPU implementation will launch a large number of CUDA threads with each thread searching a continuous section of the search space. We have decided to use the static block assignment to assign workload to CUDA threads for the GPU implementation since GPU block scheduler should be able to balance the workload (given the massive amount of blocks we are launching, the GPU scheduler basically will just act like a centralized queue).

3.5 DPLL

As mentioned earlier, DPLL uses partial assignments in order to take advantage of boolean analysis (specifically, the methods of unit clause propagation and pure literal elimination) to essentially prune the search space to look for a satisfiable assignment. While it does use some overhead in order to analyze the formula itself, the benefits far outweigh the shortcomings by significantly faster speeds compared to the naive implementation.

For specifics on the procedures, unit clause propagation refers to the scenario when there exists a clause such that all of its literals are falsified except for one, which is unassigned. In this scenario, in order for the clause and therefore the entire formula to be true, the unassigned variable must be set to true.

On the other hand, with pure literal elimination, this occurs during the scenario when we have a pure literal in the formula, which refers to when only either the positive polarity for a variable or its negation occurs in the formula (i.e either $x_i or \neg x_i$ appear in the formula). In this case, we can simply set that pure literal to true, and this should not affect the satisfiability of our formula (since the opposite of the pure literal is not there to interfere).

Again, as we stated earlier, DPLL consists of using these two procedures in between manual decisions for assigning variables with simple chronological backtracking.

3.6 CDCL

As a jump-up from DPLL, CDCL uses the decisions made in between unit clause propagations whenever an conflicting clause (i.e all of the literals in a clause of the formula are falsified by the partial assignment) occur to learn a new clause that will prevent it from making "similar" mistakes and as such make better variable decisions in future execution.

It does so by combining the decisions made at multiple levels/depths of the search tree that led to the conflicting clause to occur, and uses that information to then add another clause that will not allow the related decisions to occur again. Again, by doing so, we incur higher overhead (and now a much higher possibility for memory constraints to occur should we continuously learn new clauses without any actual

progress being made), but as we will see in the results section, this also needs to very significant speedups to both the naive and even the DPLL algorithms.

4 Results

4.1 CPU Divide and Conquer

We have implemented a sequential brute-force algorithm that iterate through the entire search space and attempt to evaluate every assignment until it either find a solution or ran out of option. This single-threaded brute-force algorithm will serve as the sequential baseline for our CPU and GPU divide and conquer algorithms. The performance of the algorithm will be measured by its execution time which is based on the wall clock time when the program starts and the wall clock time when the program finishes.

The easiest divide and conquer accomplished on CPU is a thread based approach. Each thread will process their assigned portion of the exponential search space. Out of all the work assignment, static workload assignment is the simplest and most straightforward one.

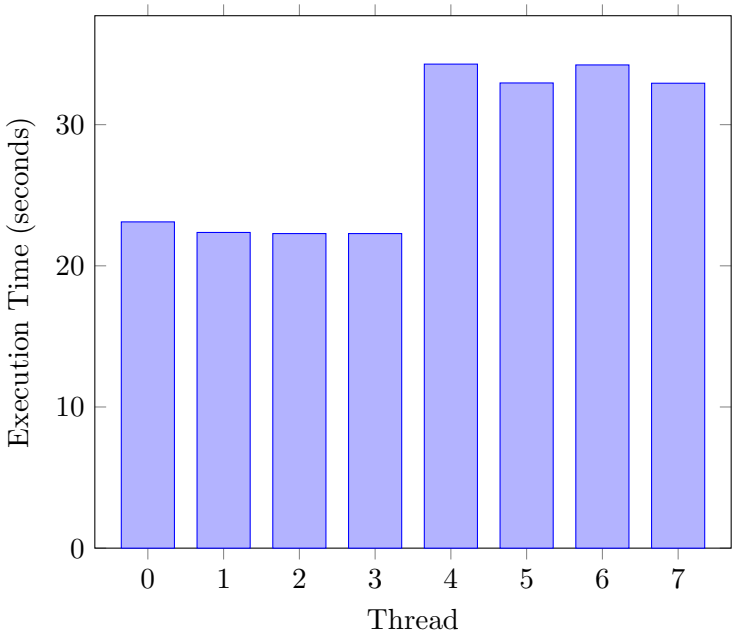
4.1.1 Block Assignment

We first start by assigning workload to each thread in block assignment. The entire exponentially sized search space is evenly divided into blocks and assigned to each thread. Every thread will evaluate the same number of boolean variable assignments in the worst case scenario. Threads communicate by a shared boolean flag. If the boolean flag is set to true by some other thread, the thread will terminate the loop early since the boolean flag indicates some other thread has solved the CNF. We were able to achieve the following result in uniformly randomly generated unsatisfiable SAT problems with 125 CNF clauses.

v_count	Sequential	2-thread	4-thread	8-thread	16-thread
27	12.2s	6.3s (1.95x)	3.2s (3.81x)	1.9s (6.42x)	1.7s (6.96x)
28	23.1s	12.0s (1.93x)	6.1s (3.79x)	3.3s (7.08x)	3.4s (6.85x)
29	49.6s	26.3s (1.89x)	14.7s (3.38x)	8.4s (5.89x)	7.7s (6.41x)
30	78.2s	41.3s (1.89x)	22.0s (3.56x)	12.6s (6.22x)	12.2s (6.43x)
31	183.8s	113.0s (1.63x)	58.8s (3.12x)	34.1s (5.39x)	30.8s (5.96x)
Avg speedup	N/A	1.858x	3.532x	6.2x	6.522x

Block workload assignment works decently well, but as shown in the benchmark, the algorithm still has room of improvement compared to a perfect speedup. We suspect that static block assignment has caused the workload to be imbalanced between threads, resulting in inefficiency in the parallel algorithm. By measuring the execution time of each thread when the program evaluated an unsatisfiable CNF clauses with 125 clauses and 31 variables, we got the following result.

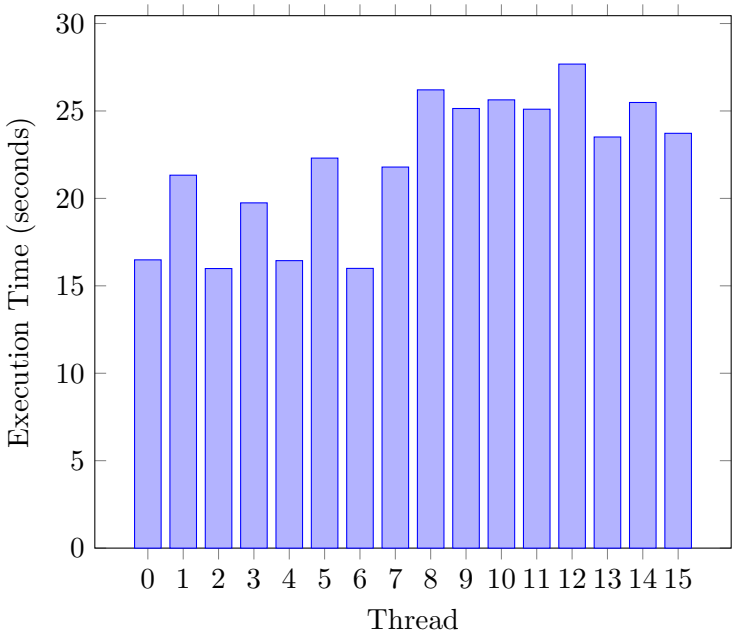
3CNF Problem with 31 Variables and 125 Clauses, Block Assignment, 8 Threads



As we could observe from the distribution of the execution time, there are threads that have been idling for $\frac{1}{3}$ of the longest executing thread's runtime. This definitely hindered us from achieving the perfect speedup.

Curiously, our program achieved better speedup when we went from 8 threads to 16 threads despite the CPU only having 8 cores and no hyper-threading. One possible explanation is that 16 threads each will have a smaller workload and the operating system could use context switch to change between threads to balance out the load on each CPU core. In other word, when we have 16 threads, the context switch from operating system is helping us to achieve a better load balance. We measure the execution time of all 16 threads when the program evaluated the same unsatisfiable CNF clauses with 125 clauses and 31 variables.

3CNF Problem with 31 Variables and 125 Clauses, Block Assignment, 16 Threads



As we could see, the execution time of thread is more evenly distributed when we have 16 threads. This

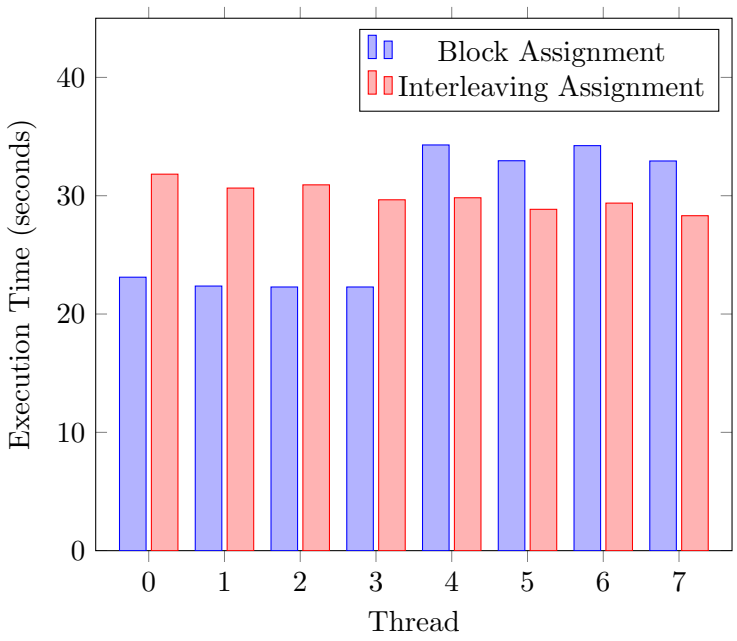
supports the theory that the context switch between threads is helping us achieving a better load balance.

4.1.2 Interleave Assignment

In block assignment, we could see that there are some threads that will finish their search much earlier than other threads. This indicates that some part of the randomly generated search space require less computation than others. We attempted interleaving assignment and see if it would help us achieving a better workload balance.

v_count	Sequential	2-thread	4-thread	8-thread	16-thread
27	12.4s	6.5s (1.89x)	3.7s (3.34x)	2.1s (5.92x)	1.8s (6.74x)
28	23.3s	13.5s (1.73x)	7.6s (3.07x)	4.0s (5.79x)	3.4s (6.8x)
29	50.3s	28.6s (1.76x)	15.1s (3.34x)	8.9s (5.66x)	8.2s (6.10x)
30	79.2s	40.9s (1.94x)	22.8s (3.47x)	13.4s (5.91x)	12.8s (6.18x)
31	203.4s	106.2s (1.92x)	58.7s (3.47x)	31.6s (6.44x)	33.3s (6.12x)
Avg speedup	N/A	1.848x	3.338x	5.944x	6.388x

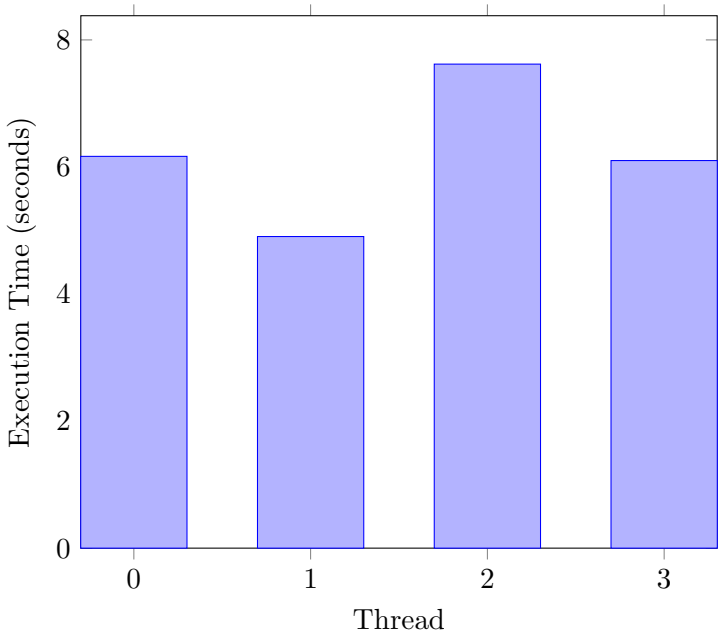
Based on the table, we could see that interleaving assignment has similar performance when compared to block assignment. In test cases where there are large skew between workload distribution among threads, we could see that the interleave assignment was able to achieve better speedup. We timed the execution time of each thread to see if the workload is more evenly distributed in those cases.



In this test case, interleaving assignment distributed the workload much more evenly amount threads. We believe that this caused the program to achieve better speedup in some test cases.

Nevertheless, interleave assignment still struggled with workload balance in a lot of other test cases such as 28 variables, 125 clauses with 4 threads as shown in the figure below. Overall, interleave assignment has slightly worse speedup when compared to block assignment. This indicate that even though interleave assignment may yield better workload balance in some cases, it will still suffer in other scenarios.

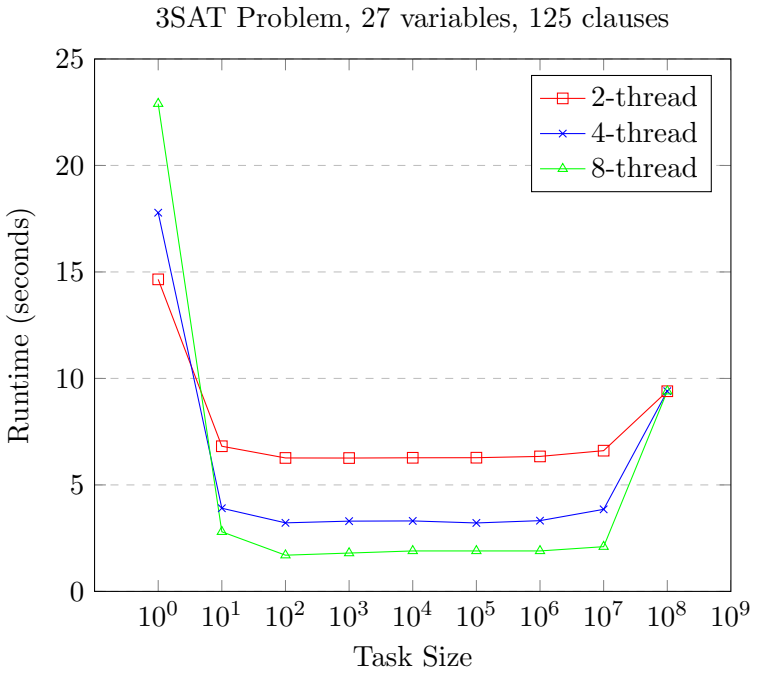
This is expected since in interleave assignment, the rightmost bits of the assignment in each thread remains fixed while in block assignment the leftmost bits of the assignment in each thread remains fixed. Essentially, there is no significant difference between interleave and block assignments.



4.1.3 Centralized Work Queue

We can see that both static workload assignment suffer from workload balance issue in different test cases. This is expected due to the random nature of SAT problems. As an effort to push for better workload balance and thereby better speedup, we have implemented a centralized work queue that supports different task size granularity to test out if we could have an even better workload balance between threads than statically assigning each thread workloads.

We first tested out different task size granularity with the small test case (27 variable 125 clauses) to determine a good granularity.



As we could see, at task size granularity of 1, the centralized queue became a point of contention and

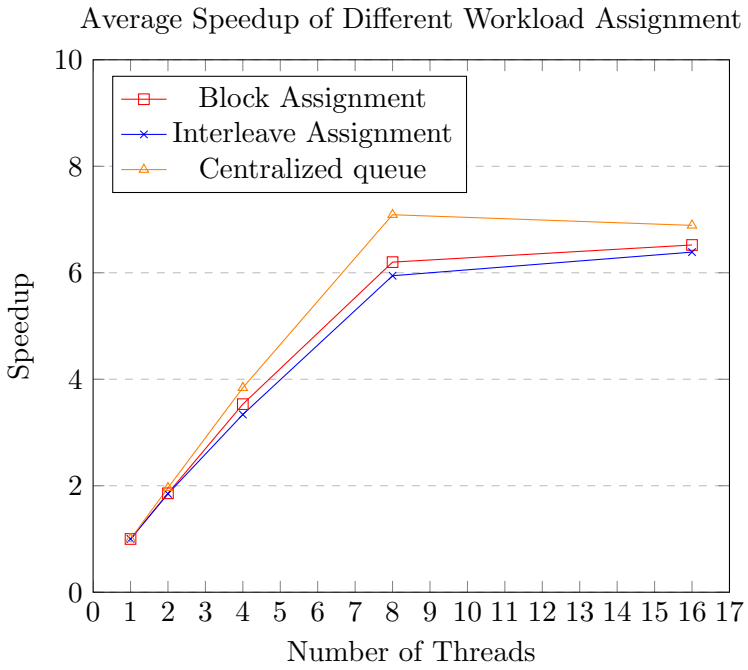
the overhead of obtaining and releasing the lock actually caused the parallel program to perform worse than the serial program. We could also see that more threads will only make this problem worse and the performance of the program actually decreases as we increase the number of threads.

However, once we increase the task size to some reasonable number, the synchronization overhead was quickly outweighed by the benefits of concurrency and the program was able to achieve a very good speedup. As the task size pass a certain threshold, it becomes so large that only 2 threads could have meaningful workload while other threads has to idle, giving us the classic U-shape associated with task granularity.

Based on this result, we have decided that 10^5 is likely a good task size granularity to use and we proceed to benchmark our centralized queue implementation with this granularity.

v_count	Sequential	2-thread	4-thread	8-thread	16-thread
27	12.3s	6.3s (1.95x)	3.2s (3.84x)	1.7s (7.34x)	1.7s (7.26x)
28	23.3s	11.9s (1.96x)	6.1s (3.85x)	3.2s (7.33x)	3.2s (7.36x)
29	50.6s	25.6s (1.98x)	13.1s (3.87x)	6.9s (7.30x)	7.2s (7.03x)
30	79.3s	40.6s (1.95x)	20.7s (3.84x)	11.5s (6.88x)	12.1s (6.57x)
31	186.7s	94.7s (1.97x)	48.9s (3.82x)	28.7s (6.50x)	29.9s (6.25x)
Avg speedup	N/A	1.96x	3.84x	7.09x	6.89x

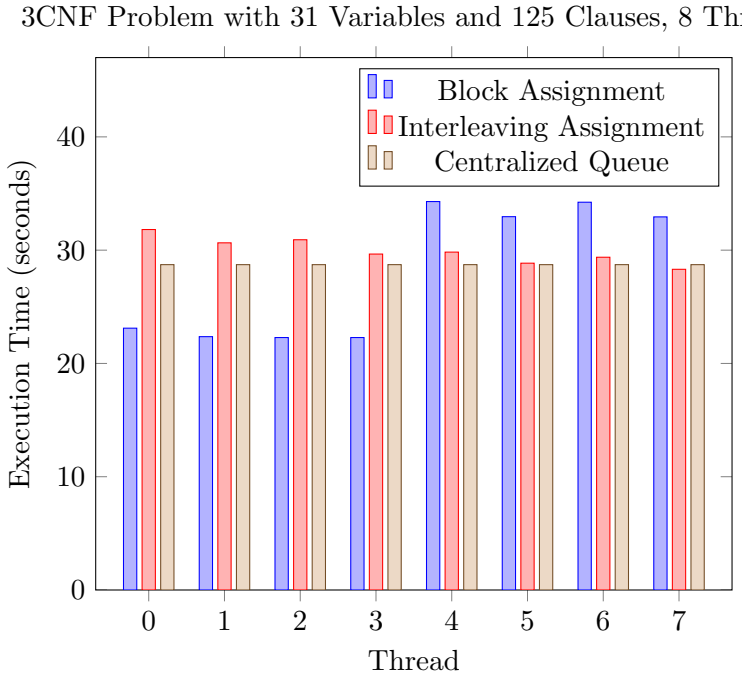
By comparing the average speedup of block assignment, interleave assignment and centralized queue, we get the following graph.



As we could see from the graph, the centralized queue approach consistently outperforms both static assignments and was able to get close to a linear speedup. Furthermore, when we increase the thread count from 8 to 16, the performance of the algorithm actually dropped when its workload assignment was determined by centralized queue. This makes sense since if centralized queue could already distribute workload evenly, adding more thread than cores would not help with workload balance and it would only increase the overhead such as context switching. Therefore, we expect that adding more threads would only decrease the performance when the workload is balanced between threads.

Under centralized queue, the difference between thread execution time ranges from 1-5ms. This indicates that the workload is pretty much perfectly balanced between threads. We believe that the incredible

workload balance generated by the centralized queue is the reason why it was able to outperform both static assignments. As we could see in the following figure, the centralized queue eliminated the variation other assignments had and was able to assign the same workload to every thread.



This consistency is observed through out all test cases. As shown in the standard deviation of thread execution time table below, we only included the standard deviation calculation when the program was executed by 8 threads because this is when the centralized queue has the most performance advantage over the other two static assignment. This is also the most relevant test case to our processor since our processor has 8 cores with no hyper-threading. The standard deviation is computed over the thread execution time of each thread in milliseconds.

v_count	27	28	29	30	31
Block Assignment	294.45	98.01	585.60	810.71	5754.43
Interleave Assignment	226.17	490.70	1440.68	902.50	1032.04
Centralized Queue	0.331	0.331	0.331	0.599	0.866

As we could see from this table, the centralized queue was able to consistently maintain an extremely low standard deviation across different test cases, indicating that every threads got the same amount of workload under centralized queue assignment in every test cases.

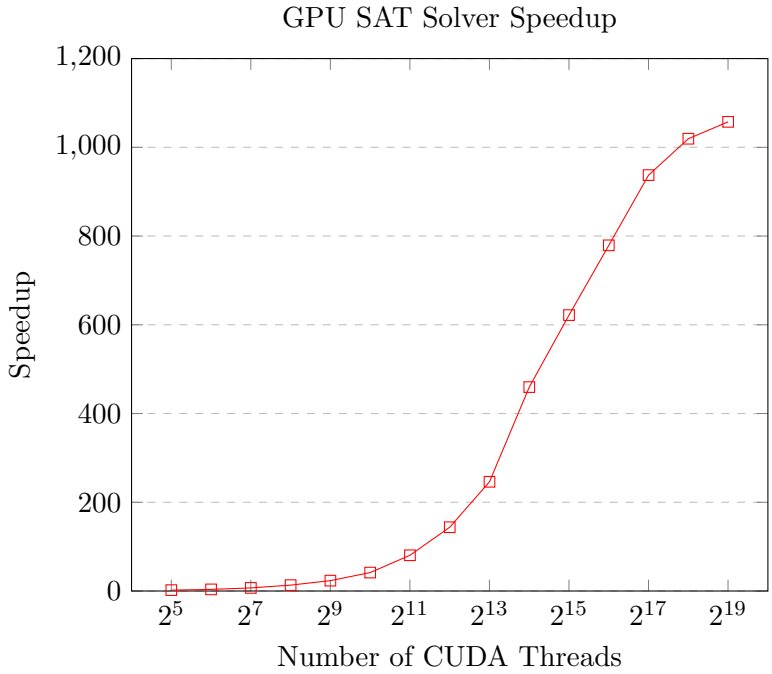
Now that we have perfectly balanced workload between threads, we believe that the synchronization cost of locking and unlocking the centralized queue, along with the overhead of creating and joining threads is limiting our program from achieving a perfect speedup. We do not believe that our program is limited by memory bandwidth or cache consistency since each thread would only read an integer vector containing 375 integers. This entire vector could easily fit inside the L1 cache of each core. Since the memory access is read only, they will not be evicted due to cache consistency issue either. This program exhibit great memory and space locality and thus memory or cache should not be the limiting factor to the speedup of the program.

By comparing all the benchmarks we have, we conclude that for CPU divide and conquer, a centralized queue workload assignment is the most optimal way to assign workload among threads.

4.2 Massively Parallel GPU Divide and Conquer

After translating the CPU thread logic into CUDA kernel, we were able to obtain the following result after various benchmarks. The GPU implementation applies the same CPU thread logic to every CUDA thread.

v_count	CPU Seq	CUDA 32-T	CUDA 64-T	CUDA 128-T	CUDA 256-T	CUDA 512-T
27	12.3s	5.9s (2.1x)	3.2s (3.9x)	1.9s (6.5x)	0.9s (12.9x)	0.5s (24.1x)
28	23.3s	11.2s (2.1x)	6.18s (3.8x)	3.55s (6.6x)	1.52s (15.3x)	1.05s (22.2x)
29	50.6s	23.8s (2.1x)	12.7s (4.0x)	6.12s (8.3x)	3.63s (13.9x)	2.00s (25.3x)
30	79.3s	47.0s (1.7x)	27.9s (2.8x)	12.2s (6.5x)	8.15s (9.7x)	4.50s (17.6x)
31	186.7s	102.9s (1.8x)	51.7s (3.6x)	26.2s (7.1x)	12.96s (14.4x)	6.78s (27.6x)
Speedup	N/A	1.96x	3.62x	7.0x	13.24x	23.36x
v_count	CPU Seq	CUDA 2^{10}-T	CUDA 2^{11}-T	CUDA 2^{12}-T	CUDA 2^{13}-T	CUDA 2^{14}-T
27	12.3s	0.32s (38.32x)	0.19s (66.49x)	0.09s (136.7x)	0.053s (232.1x)	0.029s (424.14x)
28	23.3s	0.54s (43.0x)	0.30s (78.5x)	0.15s (155.3x)	0.094s (247.9x)	0.046s (506.5x)
29	50.6s	1.13s (44.7x)	0.53s (94.8x)	0.35s (145.0x)	0.176s (287.5x)	0.089s (568.5x)
30	79.3s	2.40s (33.0x)	1.18s (67.1x)	0.67s (118.7x)	0.396s (200.3x)	0.228s (347.8x)
31	186.7s	3.83s (48.7x)	1.95s (95.9x)	1.15s (162.9x)	0.710s (262.9x)	0.414s (451.0x)
Speedup	N/A	41.54x	80.56x	143.72x	246.14x	459.6x
v_count	CPU Seq	CUDA 2^{15}-T	CUDA 2^{16}-T	CUDA 2^{17}-T	CUDA 2^{18}-T	CUDA 2^{19}-T
27	12.3s	0.020s (615x)	0.015s (820.0x)	0.013s (946.2x)	0.012s (1025.0x)	0.012s (1025.0x)
28	23.3s	0.040s (582.5x)	0.033s (706.1x)	0.034s (685.3x)	0.028s (832.1x)	0.028s (832.1x)
29	50.6s	0.069s (733.3x)	0.055s (920.0x)	0.042s (1204.8x)	0.042s (1204.8x)	0.047s (1076.6x)
30	79.3s	0.141s (562.4x)	0.111s (714.4x)	0.084s (944.1x)	0.072s (1101.4x)	0.060s (1321.7x)
31	186.7s	0.303s (616.2x)	0.254s (735.0x)	0.206s (906.3x)	0.2s (933.5x)	0.181s (1031.5x)
Speedup	N/A	621.88x	779.1x	937.36x	1019.36x	1057.38x



One thing worth noting is that 3CNF problem has extremely small size but it is very computationally demanding. 3CNF problems can usually be described within an array of integers that has less than a couple thousands elements (one 3 CNF clause contains three integers and CNF problems that we are

testing only has around one hundred clauses). This means that the problem size is around a few kilobytes. On the other hand, the computation needed to solve one problem increases exponentially. In other word, SAT problems have extremely high arithmetic intensity which makes it the perfect problem to be solved on GPU.

As we could see, after launching a massive amount of CUDA threads, we are able to achieve the targeted speedup of over 1000 times on the GPU. This indicates that the divergence problem that may negatively impact SIMD utilization did not occur or did not occur significant enough to impact the speedup.

We have also timed the data transfer time when we move the integer array representing the SAT problem from CPU memory to GPU memory and it took negligible time (0ms). This confirms that memory bandwidth is not the limit of our program and the program has extremely high arithmetic intensity. We conclude that the high arithmetic intensity of the program allowed GPU to achieve this great speedup.

The speedup from the GPU indicates that GPU is the optimal choice of machine for divide and conquer style SAT solvers.

4.3 DPLL and CDCL SAT Solver

Extending off of some of the results we got from running the naive implementation and some of the CUDA-threaded implementation, we display DPLL and CDCL runtime and speedups on those same inputs as well as much larger inputs much better suited for the algorithms.

v_count	CPU Seq	CUDA 2^{12}-T	CUDA 2^{19}-T	DPLL	CDCL
27	12.300s	0.090s (136.7x)	0.012s (1025.0x)	0.001s (12,300x)	0.0006s (20,500x)
31	186.700s	1.150s (162.9x)	0.181s (1031.5x)	0.0016s (116,688x)	0.0008s (233,375x)
50	3 years?	N/A	N/A	0.0021s	0.0009s
125	Heat Death?	N/A	N/A	0.210s	0.0017s
250	Eternal Silence	N/A	N/A	17.252s	0.0027s
30000	Universe 2.0	N/A	N/A	N/A	10.332s

First, we again note the impressive speedups that the CUDA implementations were able to achieve relative to the CPU. Alone, it shows the powerful capabilities for compute-based hardware such as a GPU when used correctly / in the correct context.

However, when factoring in the results from DPLL and CDCL, it still shows that algorithmic improvements when facing common problems are just as important as working on highly-capable hardware when performing tasks. We note that although these were somewhat optimized instances of CDCL and DPLL, there were still many state-of-the-art modifications that could be made to these intial implementations to further improve both the memory constraints and the run-times for these algorithms. Additionally, from some CMU lecture slides, we have that CDCL is notoriously hard to parallelize [4], but there is nevertheless continuous research going on in trying to optimize CDCL both parallel-wise and algorithm-wise.

(Additionally, due to the blowing-up nature of the CPU sequential timings, we have added alternative timing measurements for the reader's enjoyment.)

4.4 Portfolio SAT Solver

To recap, the crude portfolio method for parallelizing a SAT-solver involves running different algorithms or variations of an algorithm separately (where each instance takes a core for itself). In our case, we have chosen the variation to be how we decide which variable to decide when we have to do a manual decision. The six options we test (denoted v1 to v6, and they are given in choose.cpp) are as follows:

1. Choose the literal with the highest frequency.

2. Choose the negation of the literal with the high frequency.
3. Choose the variable (combine counts for positive and negative literal) with the highest frequency, and then pick the more frequent of the two literals that can be made from that variable (positive or negative).
4. Choose the variable (combine counts for positive and negative literal) with the highest frequency, and then pick the less frequent of the two literals that can be made from that variable (positive or negative).
5. Randomly choose a variable, and then pick the more frequent of the two literals that can be made from that variable (positive or negative).
6. Pick the first unassigned variable in order ascending from one, and then pick the more frequent of the two literals that can be made from that variable (positive or negative).

With these six variations (done using CDCL), we test them on some inputs and get the following results:

Variation	6000v3SAT	30000v3SAT	Blocks3016v	Blocks6325v
V1	48ms	10243ms	252ms	1090ms
V2	42ms	8553ms	283ms	948ms
V3	49ms	11202ms	283ms	1621ms
V4	44ms	9339ms	295ms	1673ms
V5	48ms	10565ms	316ms	1189ms
V6	50ms	10779ms	222ms	998ms
Min	42ms	11202ms	222ms	948ms
Max	50ms	8553ms	316ms	1673ms

where the two problem types on the left correspond to 6000/30000 variable 3SAT problems and the two problem types on the right correspond to SAT-encodings for the Blocks World planning problem (commonly used in AI scenarios) which were taken from the UBC SATLib website [3].

In terms of how these values correspond to the portfolio method, the portfolio method would return / finish in the minimum time given for each of the corresponding columns. Compared to the maximums (which could be considered as our pseudo-worst-case scenario), we do notice some noticable differences between the two.

Additionally, we note that V2 seemed to generally be strong in its runtime compared to the others; I did not necessarily expect this to be the case as I expected the opposite of it (V1) to be the strongest since it would set the most amount of clauses to true compared to the other heuristics, which should speed up our searching. However, with V2, since it is setting the most amount of terms to be false, this may allow for more unit clauses to appear (since clauses need all but one of their literals to be falsified) which therefore may allow for better unit propagation which may result better learned clauses. We note that this is also ongoing study in this area, which many complex heuristics (including those using deep learning as a scoring function to determine which variable to set) which seem promising.

5 Credit

Overall we believe that the team had a 50%/50% split in the work done with Jerry focusing on parallelizing divide and conquer algorithms and Tejas focusing on implementing advanced sequential algorithms.

Jerry

- Implementing python uniformly random 3SAT problem generator.

- Implementing 3SAT python solution verifier.
- Implementing and benchmark all divide and conquer algorithms
  - static block workload assignment
  - static interleave workload assignment
  - dynamic centralized queue workload assignment
  - GPU Divide and Conquer SAT solver

Tejas

- Implementing initial and final optimized versions of the fast, sequential SAT solving algorithms (DPLL and CDCL approaches).
- Benchmarking the optimized fast, single-threaded algorithms
- Benchmarking the potential/observed portfolio speedups with various heuristics

References

[1] "Boolean Satisfiability Problem." Wikipedia. Wikimedia Foundation, October 24, 2022. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[2] "SAT Solver." Wikipedia. Wikimedia Foundation, September 1, 2022. https://en.wikipedia.org/wiki/SAT_solver.

[3] SATLIB - Benchmark Problems. https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[4] Carnegie Mellon University - Logic and Mechanized Reasoning Conflict-Driven Clause-Learning Solving. https://www.cs.cmu.edu/~mheule/15217-f21/slides/cdcl.pdf