Solving the Independent Set Problem using the Exhaustive Search algorithm

Kshitij Agarwal, Aurodeep Ghosh, Reda Elbahri

Project Mentor: Prof Alan Kaminsky

Description of the Independent Set Problem

The independent set problem is an NP problem related to the graph theory.

The input is a graph containing V vertices and E edges. We need to find a subset of vertices S, in the given graph, such that no two vertices are adjacent in S, that is, there is no edge directly connecting any two vertices in the set S. The set S is then called an independent set and the number of vertices in this set is the size of this independent set.

The exhaustive search algorithm is about finding such an independent set, given the size of the independent set as an input, that is, if the input is an integer I, then we have to find an independent set of size I in a particular graph, which is also given as an input or it can be a randomly generated graph as well.

Exhaustive Search algorithm for the Independent Set Problem

An exhaustive search algorithm can be understood as a brute force search that tries all the possible inputs to find the solution to the given problem.

For the independent set problem, we need to find all the possible inputs to the algorithm, that is, we need to find all the possible subsets from the given set of vertices V of the given size I. Every time a subset is generated, it is checked that whether it is an independent set or not, that is, it is checked whether is it true that there is no edge between any two chosen vertices within that generated subset or not. Every possible combination is tried to arrive at the correct answer.
Description of the algorithm from the first research paper

Implementation of Maximum Independent Set Problem by Algorithmic Tile Self-assembly

In this paper, the authors brought the light on utilizing the parallelism in computing to support founding solution to the DNA tile algorithmic self-assembly problem as an N-P problem. The structure of the problem shows a direct proportion of the required computations with the size of the problem, whereas the size of the DNA tile problem is $O(n)$ and the total number of iterations is $O(mn)$, it is obvious that a maximum independent set approach is considered as an efficient application to this problem.

Using DNA tile self-assembly to solve the NP problem is achieved through three systems:

1) Non-deterministic guess system.

It is the process of checking the inequality characteristic among elements of a set of tiles ($1..n$) by non-deterministically guessing the value of subset of tiles from our original set and setting those vertices to 0 or 1 only according to the type of tiles and create the adjacency matrix for those tiles. The rest of the values for the original set are generated by passing the newly computed values of the subset to the upper tiles and distinguishably marked to apply AND operation next.

2) AND operation system:

A logical operations AND is performed between the coefficient of the vertices and their computed values in the guessing system. Basically, the tile contains 5 pockets of which express the results of the operations and the computed values. For instance, the first row of tiles, represent the vertex 1 vector, has left, right, and center pocket. The bottom contains the value of the vertex where the left contains the corresponding coefficient coordinate in the coefficient matrix. The result of AND operation between the right and bottom is placed in the center pocket. The value of the right and bottom has to be passed to the left and upper pockets respectively. The second row perform the same routine of copying the values in the corresponding pockets and passes the values to the upper layer in order to compute new values.

The bottom pocket in the third and fourth tile are the vertices values and they should be passed to the upper layers in order to perform the comparison in the comparison system.

3) The comparison system:

Basically, the system performs comparisons to conclude the inequality of the vertices and investigating in whether the vertices are adjacent. The right pocket is checked, after subtraction between the ANDed value and the coefficient value in the right pocket, and the left pocket will carry the result of the compared value with the representation of ($<=$0) in case of the operation
result has been 1 and the subtracted coefficient value was 1 and result of this comparison means that the value of the left pocket is can't be bigger that the right. On the other hand, if the result of AND operation was 0 and the and the right pocket contains the value of 1, then the relationship is (<=1) and denotes that the inequality is characterized for such vertex.

The paper also pointed to the comparison of the last vertex \(v_n\) and other further layers of the matrix of tiles of which can be computed by applying the comparisons by inferring the resulted computed values for preceding operations and assigning the value based on that to perform the comparison.
Parallel Computing I Spring 2013 - Team Project Report

Description of the algorithm from the second research paper

Parallel Algorithms for Finding a Near-Maximum Independent Set of a Circle Graph

The paper is about searching for the maximum independent set with the largest cardinality among other independent sets in a graph of vertices and edges. The methodology of this algorithm is best suited to solve the problem of finding the secondary structure in RNA (ribonucleic acids) by iterating the computation for several hundreds of times. The simulator is built on the algorithm that runs \( n \) processing elements, it is called the McCulloch-Pitts binary neuron, and for every processing element, the computation is performed as a binary function of:

\[ V_i = 1 \text{ if } U_i > 0 , \text{ output } 0 \text{ else output is } V_i \text{ and } U_i \text{ are the output of the function and the input for the next iteration.} \]

The complexity of the problem is an N-P problem with \( O(n^2) \). In fact, the motion equation to compute Mc-Culloch-Pitts neuron is stated in the formula:

\[
\Delta U_i (t) = A \left( \sum_{j=1}^{n} d_{ij} (1 - V_j (t)) \left( \text{distance} (i) \right)^{-1} \right) \\
\cdot (1 - V_i (t)) \\
- Bh \left( \sum_{j=1}^{n} d_{ij} (1 - V_j (t)) \right) V_i (t).
\]

Motion Equation for the Mc- Culloch-Pitts binary neuron

Where \( d_{ij} \) is the Euclidean distance between the tail and the head of the neuron in the graph planar and \( h(x) \) is 1 if \( x = 0 \) and 0 otherwise. The factor \( A \) and \( B \) are represented in the experiment by value of 1. The edges are stated to be non-intersected, if and only if for every head in edge \( i \), both the head and tail, the vertices of the edge, of the edge \( j \) are not equal and the head is bigger than the tail of the individual edge.

Algorithm Notion:

1) Preset the time \( t \) to be 0 as an initial value
2) \( U_i \) for \( t = 0 \) is sat to negative or random number
3) Compute \( V_i \), as a part of the algorithm the binary function states for every \( V_i(t) = 1 \) the edge is not embedded in the circle graph, in other words the edge is intersected with the other edge/s in the graph, and the output of the neuron to the motion equation is greater than the value of zero.
4) Compute \( \Delta U_i (t) \) in the formula above.
5) Compute $U_i(t + 1)$ to by applying the first-order Euler method and go to last step
6) Compute the $\Delta U_i(t)$ after incrementing the value of t to $(t +1)$. If $\Delta U_i(t+1) = 0$, iteration terminates and return to step 3.

The authors pointed to a previous trial to solve such a problem using back-propagation algorithm using a three-layer feed-forward neural network [5]. In this approach, the algorithm had to learn about find the proposed correlation between the amino acid sequence and the secondary basis structure, the approach also suffered from the lower accuracy of the learning method and feed-forward of the neural network was unable to catch up with the prediction for the long sequences of the second structure.

The algorithm in this paper, even it runs currently on a sequential environment, is able to traverse every edge and maximize the results by finding the all possible edges in the circular graph that are not embedded neurons (intersected edges).

The algorithm was tested to solve the problem of new structures in a sequence of 38 bases and 152 edges, 55 bases and 331 edges as well as 359 bases and 1017 edges from the potato spindle tuber viroid (PSTV). The problem was stated to calculate $m$ vertices with $n$ edges, 38 vertices and 152 edges for the first case study, that represent the number of pair bases in the second structure. It was obvious from the experiment that the convergence around the factor of 14 embedded edge was factual since even by the substantial jumps in the number of the iterations, the number remained stable, with a coefficient $A = 1$ and $B \geq 0.2$ and it dwindle after that.

The other cases showed symmetrical behavior where the convergence was measured at the level of 500 iterations in general with multiple measured corresponding coefficients and stability factors.
Description of the algorithm from the third research paper

A simple parallel algorithm for the maximal independent set problem

This paper discusses simple parallel algorithms for the Maximum Independent Set problem based upon the Monte Carlo algorithm and its derivatives by converting them into deterministic models.

The Maximum Independent Set in an undirected graph describes the maximum collection of vertices from the graph which are not connected as a pair. The first algorithm is based upon the local property which takes any random vertex into consideration and adds it to the set if it is not connected to any of its adjacent vertices. So pairwise independence is the driving factor for this configuration. This local property is basically the conversion of mutually independent events in the large sample space to pairwise independent events in comparatively smaller sample space. The second derivative algorithm uses the concept of randomly generated bits to choose the smaller sample space from the larger one and the third algorithm does this in parallel, picking up the vertices which confirm to pairwise independence in separate steps making it deterministic.

The test bed is an EREW P-RAM parallel computer with non-concurrent read-write cycles for the same memory locations. Below are the algorithm characteristics as mentioned. ‘EO’ depicts the expected value of complexity and since the third algorithm is deterministic in nature, the time complexity is not expected in form.

<table>
<thead>
<tr>
<th>Algorithm - #</th>
<th>Processors Used</th>
<th>Time Complexity</th>
<th>Random Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(m)$</td>
<td>$EO((\log n)^2)$</td>
<td>$EO(n\log n)$</td>
</tr>
<tr>
<td>2</td>
<td>$O(m)$</td>
<td>$EO((\log n)^2)$</td>
<td>$EO((\log n)^2)$</td>
</tr>
<tr>
<td>3</td>
<td>$O(n^2m)$</td>
<td>$O((\log n)^2)$</td>
<td>none</td>
</tr>
</tbody>
</table>

Looking closely, we see that the first algorithm accepts an undirected graph $G = (V, E)$ as input and the output is a maximal independent set. Being an iterative algorithm, the while loop execution takes $O((\log n)^2)$ time as we see below in two steps.

BEGIN
I = NULL
G’ = (V’, E’) - G = (V, E)
while V’ != NULL
    X = NULL
    In parallel, for all $v \in V’$,
    (*) Randomly choose to add $v$ to $X$ with probability $1/(2.d(v))$
Comparing the number of edges in set $G'$ before and after the execution of the while loop confirms the run time complexity and in depth discussion is beyond the scope of the paper.

The first algorithm ran on the precincts that the problem space comprised of mutually independent events, but the next two variations are based on the condition that the problem space now comprises of vertices that are pairwise independent. This is based upon a few assumptions, first that the problem space comprises only of binary vectors. The generation of a pairwise independent set from mutually independent set is done using two lemmas which show that the sample space has a multiple of $q^2$ vectors.

1. \[ \text{Pr}[E_i] = p_i' \]
   for fixed $l$, there are exactly $q$ pairs of $x, y$ such that $(x+y . i) \mod q = l$. Event $E_i$ occurs in all $b_{x, y}$ such that $(x+y . i) \mod q$ is between 0 and $p_i' \cdot (q - 1)$, inclusive, i.e. for exactly $p_i' \cdot q^2$ boolean vectors.

2. \[ \text{Pr}[E_{i1} \cap E_{i2}] = p_{i1}' \cdot p_{i2}' \]
   for fixed $l_1$ and $l_2$, there are exactly one $x, y$ pair such that $(x+y . i_1) \mod q = l_1$ and $(x+y . i_2) \mod q = l_2$ simultaneously. Event $E_{i1}$ and $E_{i2}$ both occur for $p_{i1}' \cdot p_{i2}' \cdot q^2$ pairs of $l_1, l_2$. Thus $E_{i1}$ and $E_{i2}$ occur for $p_{i1}' \cdot p_{i2}' \cdot q^2$ boolean vectors.

The algorithm uses this simplification to simulate the problem in a parallel architecture. With only $q^2$ binary vectors in consideration, the deterministic algorithm runs in a sample space of only $O(n^2)$. Since the probabilities in the new space don’t actually match the probabilities of the original sample space, there is a small alteration to the algorithm. If there exists a certain vertex whose degree $d(v) \geq n/16$, the vertex is immediately added to the independent set and it removed from the actual graph. We see this in the algorithm implementation, using $O(m)$ processors, a while loop runs in $O(\log n)$ and there will be $O(\log n)$ while loops leading to the combined time complexity $O((\log n)^2)$. But the third variation, leaves out the random choices made in the first variation (*).
BEGIN
I = NULL
G' = (V',E') - G = (V, E)
while V' != NULL
    BEGIN
        In parallel, for all v Є V', compute d(v)
        In parallel, for all v Є V',
        If d(v) =0 then always add v to I and delete v from V'
        compute n' = |V'|
        find v Є V' such that d(v) is maximum
        if d(v) >= n'/16 then add v to I and let G' be the graph
            induced on the vertices V' − ((v) U N(v))
        else ( for all v Є V', d(v) < n'/16)
            BEGIN
                compute a prime q such that n' <= q <= 2n'
                randomly choose x and y such that, 0 <= x, y <= q - 1
                X = NULL
                In parallel, for all v Є V',
                    BEGIN
                        compute n(v) = ⌊(q / 2d(v))⌋
                        compute l(v) = (I + v.j) mod q
                        if l(v) <= n(v) then add v to X
                    END
                I' = X
                In parallel, for all v Є X, w Є X.
                if (v, w) Є E' then
                    if d(v) <= d(w) then I' = I' − {v}
                    else I' = I' − {w}
                I = (I U I')
                Y = I' U N(I')
                G' = (V'.E') is the induced subgraph on V' - Y.
            END
        END
    END
END

So the algorithm cycles through all q^2 pairs in parallel. But an addition to this scenario would be an increase in the number of processors used for the run. So instead of O(m) processors initially, O(n^2 m) processors are used now. Now the running time becomes a deterministic O((log n)^2).
A high level design of the sequential program

<table>
<thead>
<tr>
<th>IndependentSetSeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectivitiyMatrix : int ([][])</td>
</tr>
<tr>
<td>I : int</td>
</tr>
<tr>
<td>independentSubSets : List&lt;Set&lt;Integer&gt;&gt;</td>
</tr>
<tr>
<td>noOfIndependentSubSets : int</td>
</tr>
<tr>
<td>noOfPossibleSubsets : int</td>
</tr>
<tr>
<td>seed : long</td>
</tr>
<tr>
<td>V : int</td>
</tr>
<tr>
<td>findIndependentSets() : void</td>
</tr>
<tr>
<td>generateMatrix() : void</td>
</tr>
<tr>
<td>isIndependentSet(Set&lt;Integer&gt;) : boolean</td>
</tr>
<tr>
<td>main(String[]) : void</td>
</tr>
<tr>
<td>printMatrix() : void</td>
</tr>
<tr>
<td>usage() : void</td>
</tr>
</tbody>
</table>

The data members are described as follows(mentioned in the order as written in the program):

**User Inputs :-**

V : Stores the number of vertices that will be present in the randomly generated undirected graph. It is an integer, taken as an input from the user.

I : Stores the size of the independent set. It is an integer, taken as an input from the user. This tells the program, the size of the independent set that has to be found, from the randomly generated graph.

seed : The pseudo random number generator seed.

**User input just for the main program:-**

PrintMatrix: Tells the program whether to print the randomly generated connectivity matrix as output or not. If 'I' is given as input, the matrix is printed, otherwise it is not printed.
Non-user Inputs :-

**connectivityMatrix** : Stores the randomly generated undirected graph information in the form of adjacency matrix. It is a double dimensional array in which a particular element at \(i^{th}\) row and \(j^{th}\) column tells that whether there exists an edge between the two vertices numbered as \(i\) and \(j\), or not. The number '1' represents that there is an edge and '0' represents that there isn't an edge.

**noOfPossibleSubsets** : Stores the number of total possible subsets, computed as \(2^V\) after \(V\) is received from the user. It is utilized to loop over all the possible subsets, where the subsets of size \(I\) are processed and the rest of them are ignored.

**independentSubSets** : Stores the calculated independent sets of the given size \(I\). It is a list which stores the sets containing the identifiers of the vertices. Each set contains \(I\) elements, representing \(I\) vertices, which make an independent set.

**noOfIndependentSubSets** : Stores the number of independent sets of the given size \(I\) that are found in the randomly generated undirected graph. It is an integer variable.

Working of the sequential program:-

**Pseudo random generation of the connectivity matrix** - The maximum number of the edges a simply connected undirected graph can have is \((V*(V - 1) / 2)\), where \(V\) is the total number of vertices in the graph. The number of edges that are generated are more than half of this maximum number of edges. This number is generated randomly using a pseudo random number generator. The generator keeps generating the value for the number of edges, until finally the number generated is greater than the half of \((V*(V - 1) / 2)\).

**Finding the independent sets** - The total number of possible subsets of vertices \(T\), for a given number of vertices \(V\), is \(2^V\). Ignoring the null set, a loop iterates from 1 to \(T\), looking over the binary representation of all the numbers represented by the loop index. If the loop index integer is written in two's complementary binary representation, and the one bit represents a particular vertex that is in the subset and the zero bit represents a particular vertex that is not in the subset, then the number of one-bits will tell us that whether that particular subset will be of size \(I\) or not.

Accordingly, if we acquire a binary representation of the loop index which can give us a subset of size \(I\), the program proceeds further with the generation of the subset, otherwise it proceeds with the next loop index. After the subset is generated, it is fed to a subroutine that checks whether it is an independent set or not. If yes, the subroutine returns true, the subset is added to the **independentSubSets** list and **noOfIndependentSubSets** is incremented by one. Both of these are specified in the data structure above.

After all the independent sets are collected, all of them are printed along with the total number of independent sets found.
A high level design of the parallel program

```
<table>
<thead>
<tr>
<th>IndependentSetSmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectivityMatrix : int ([][])</td>
</tr>
<tr>
<td>I : int</td>
</tr>
<tr>
<td>independentSubSets : List&lt;Set&lt;Integer&gt;&gt;</td>
</tr>
<tr>
<td>noOfIndependentSubSets : SharedInteger</td>
</tr>
<tr>
<td>noOfPossibleSubsets : int</td>
</tr>
<tr>
<td>seed : long</td>
</tr>
<tr>
<td>V : int</td>
</tr>
</tbody>
</table>

+ findIndependentSets() : void |
+ generateMatrix() : void |
- isIndependentSet(Set<Integer>) : boolean |
+ main(String[]) : void |
- printMatrix() : void |
- usage() : void |
```

The data members are described as follows (mentioned in the order as written in the program):

**User Inputs :-**

They are the same as described for the sequential program above.

**Non-user Inputs :-**

- **connectivityMatrix** : Same as described for the sequential program above.
- **noOfPossibleSubsets** : Same as described for the sequential program above.

- **independentSubSets** : Stores the calculated independent sets of the given size I. It is a thread-safe list which stores the sets containing the identifiers of the vertices. Each set contains I elements, representing I vertices, which make an independent set.
- **noOfIndependentSubSets** : Stores the number of independent sets of the given size I that are found in the randomly generated undirected graph. It is a thread-safe integer variable.
Working of the parallel program:-

**Pseudo random generation of the connectivity matrix** - It is same as described in the working of the sequential program above.

**Finding the independent sets** - It is same as described in the working of the sequential program above except for the following changes. The loop iterating over 1 to T (total number of possible subsets of vertices) is divided over the number of processors available. Each processor is supplied with different possibilities of choices as inputs to the exhaustive search algorithm in the form of different loop indexes to work with. Also, the results are calculated in the same way, but now the list in which the independent sets are stored has been made thread safe. The number of independent subsets are also recorded in the thread safe integer variable. These are made because multiple thread might be willing to access the list and the integer variable at the same time.

**Load Balancing** - It is not necessary that if the loop iterations from 1 to T (total number of possible subsets of vertices) are split amongst the available processors, the running time of all the threads will be the same. It comes from the fact that in each chunk of iterations, it is not necessary that there will be the same number of subsets of size I. Also, the calculation time of checking an independent set also varies, and is not the same for all the sets that will be checked. Thus, a guided schedule is used for balancing the load.

**Note** - The printout of the resulting independent sets in the cases of sequential and parallel programs will be different, but the total number of solutions found remains the same.
The developer's manual for the software

Download and install:-

RIT's Parallel Java Library
http://www.cs.rit.edu/~ark/pj.shtml

- Extract the zip file containing all the programs to a directory of your choice
- Set the class path first to the current directory and then to the parallel java library.
- Compile all the programs after the above steps by javac *.java

To run the program on RIT CS Parasite 8-Core SMP
- Compile all the programs by javac -source 1.5 -target 1.5 *.java

Please refer to the following information for the version of java
http://www.cs.rit.edu/~ark/runningpj.shtml#javaversion
For more information, please refer the following:-
http://www.cs.rit.edu/~ark/runningpj.shtml

The user's manual for the software

Sequential Version
java IndependentSetSeq <V> <I> <seed> <PrintMatrix>
where:-
<V> = The total number of vertices in the graph
<I> = The size of the independent set
<seed> = The pseudo random number generator seed
<PrintMatrix> = 1 for yes, 0 for no

SMP Parallel Version
java -Dpj.nt=<NT> IndependentSetSmp <V> <I> <seed> <PrintMatrix>
where:-
<NT> = The number of parallel threads
<V> = The total number of vertices in the graph
<I> = The size of the independent set
<seed> = The pseudo random number generator seed
<PrintMatrix> = 1 for yes, 0 for no
Performance Metrics measured for the parallel program

Notations for the performance metrics:-
N : Problem size
T : Running time
K: Number of processors

The performance metrics are the speedup of the program as following:-
1.) The ratio of the running time of the sequential program with respect to the running time of the parallel program.
\[
\text{Speedup}(N,K) = \frac{T_{\text{seq}}(N,1)}{T_{\text{par}}(N,K)}
\]
2.) Efficiency = Speedup(N,K)/K

Notations for the Output Tables:-
NT = The number of parallel threads
T1 = Running time for first program run
T2 = Running time for second program run
T3 = Running time for third program run
T = Smallest Running time amongst T1, T2 and T3.

Speedup and efficiency are measured on the basis of the smallest running times.

Note- All the readings have been taken with PrintMatrix as 0. PrintMatrix feature is provided only to verify the output for small instance of the independent set problem.
The running times have been measured on the parasite.cs.rit.edu SMP computer.

Input 1:
java IndependentSetSeq 100 5 423345 0
java -Dpj.nt=1 IndependentSetSmp 100 5 423345 0
java -Dpj.nt=2 IndependentSetSmp 100 5 423345 0
java -Dpj.nt=3 IndependentSetSmp 100 5 423345 0
java -Dpj.nt=4 IndependentSetSmp 100 5 423345 0
java -Dpj.nt=8 IndependentSetSmp 100 5 423345 0

Output for Input 1:

<table>
<thead>
<tr>
<th>NT</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>56016</td>
<td>55946</td>
<td>42340</td>
<td>42340</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>51815</td>
<td>51656</td>
<td>51649</td>
<td>51649</td>
<td>0.81976</td>
<td>0.81976</td>
</tr>
<tr>
<td>2</td>
<td>32095</td>
<td>29179</td>
<td>31287</td>
<td>29179</td>
<td>1.45104</td>
<td>0.72552</td>
</tr>
<tr>
<td>3</td>
<td>18897</td>
<td>16650</td>
<td>21234</td>
<td>16650</td>
<td>2.54294</td>
<td>0.84765</td>
</tr>
<tr>
<td>4</td>
<td>15582</td>
<td>19504</td>
<td>18031</td>
<td>15582</td>
<td>2.71724</td>
<td>0.67931</td>
</tr>
<tr>
<td>8</td>
<td>10351</td>
<td>9796</td>
<td>12184</td>
<td>9796</td>
<td>4.32217</td>
<td>0.54027</td>
</tr>
</tbody>
</table>
**Input 2:**
java IndependentSetSeq 50 5 423345 0
java -Dpj.nt=1 IndependentSetSmp 50 5 423345 0
java -Dpj.nt=2 IndependentSetSmp 50 5 423345 0
java -Dpj.nt=3 IndependentSetSmp 50 5 423345 0
java -Dpj.nt=4 IndependentSetSmp 50 5 423345 0
java -Dpj.nt=8 IndependentSetSmp 50 5 423345 0

**Output for Input 2:**

<table>
<thead>
<tr>
<th>NT</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>53205</td>
<td>55410</td>
<td>59727</td>
<td>53205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>49352</td>
<td>49253</td>
<td>55494</td>
<td>49253</td>
<td>1.08024</td>
<td>1.08024</td>
</tr>
<tr>
<td>2</td>
<td>26281</td>
<td>29999</td>
<td>27305</td>
<td>26281</td>
<td>2.02447</td>
<td>1.01223</td>
</tr>
<tr>
<td>3</td>
<td>15639</td>
<td>18395</td>
<td>15673</td>
<td>15639</td>
<td>3.40207</td>
<td>1.13402</td>
</tr>
<tr>
<td>4</td>
<td>15104</td>
<td>15078</td>
<td>15094</td>
<td>15078</td>
<td>3.52865</td>
<td>0.88216</td>
</tr>
<tr>
<td>8</td>
<td>7604</td>
<td>7074</td>
<td>7239</td>
<td>7074</td>
<td>7.5212</td>
<td>0.94015</td>
</tr>
</tbody>
</table>

**Input 3:**
java IndependentSetSeq 125 5 423345 0
java -Dpj.nt=1 IndependentSetSmp 125 5 423345 0
java -Dpj.nt=2 IndependentSetSmp 125 5 423345 0
java -Dpj.nt=3 IndependentSetSmp 125 5 423345 0
java -Dpj.nt=4 IndependentSetSmp 125 5 423345 0
java -Dpj.nt=8 IndependentSetSmp 125 5 423345 0

**Output for Input 3:**

<table>
<thead>
<tr>
<th>NT</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>55371</td>
<td>41462</td>
<td>55104</td>
<td>41462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50811</td>
<td>53994</td>
<td>50974</td>
<td>50811</td>
<td>0.816</td>
<td>0.816</td>
</tr>
<tr>
<td>2</td>
<td>27794</td>
<td>29221</td>
<td>28307</td>
<td>27794</td>
<td>1.49176</td>
<td>0.74588</td>
</tr>
<tr>
<td>3</td>
<td>18019</td>
<td>18105</td>
<td>18071</td>
<td>18019</td>
<td>2.30102</td>
<td>0.76701</td>
</tr>
<tr>
<td>4</td>
<td>14649</td>
<td>14582</td>
<td>14646</td>
<td>14582</td>
<td>2.84337</td>
<td>0.71084</td>
</tr>
<tr>
<td>8</td>
<td>9555</td>
<td>8499</td>
<td>9089</td>
<td>8499</td>
<td>4.87846</td>
<td>0.60981</td>
</tr>
</tbody>
</table>
A discussion of what we learned from the project

The project provided insights into the creation of a sequential program design for a given NP problem (Independent Set), analyzing the performance and the methods of converting it into an optimized parallel design to generate speed ups. At a smaller scale, an independent set problem might be simpler to solve, but upon increasing the size of the graph in terms of vertices poses a huge complexity in terms of computational time and space. Present day computation systems boast of multi-core architectures. This project provided us a chance to learn how to optimize code designs for parallel architectures as high performance scientific computing problems like any NP problem, generally are massive and demand comparatively larger processing power. So it was wise to design our code based upon it.

The Parallel Java Library, developed by Prof. Alan Kaminsky, made it easy to design the parallel programs in Java. The SMP program developed using the Parallel Java Library, performs much better in terms of speed up and helped us solve the independent set problem in lesser time. Also, the research papers studied during the course of the project provided us a deeper insight into the various methods of solving such a NP problem using parallel computing.

A discussion of the possible future work

A cluster version of the same program can also be made, which can be run on the cluster parallel computer like RIT CS Paranoia 32-Processor Cluster.

Furthermore, an analysis can be made that whether this problem is a good candidate for a GPU parallel program or not. If the analysis says yes, then a GPU parallel program can be made for it.

A statement of what each individual member did on the project

Initially, Aurodeep Ghosh developed the recursive sequential version of the program which was converted to an iterative version by Kshitij Agarwal. Also, Kshitij Agarwal developed the parallel version of the program and the team deliverable report.

The analysis of research papers and the description of the algorithms from the research paper were done by Reda Elbahi and Aurodeep Ghosh.
References


6.) Parallel Java Library developed by Prof. Alan Kaminsky