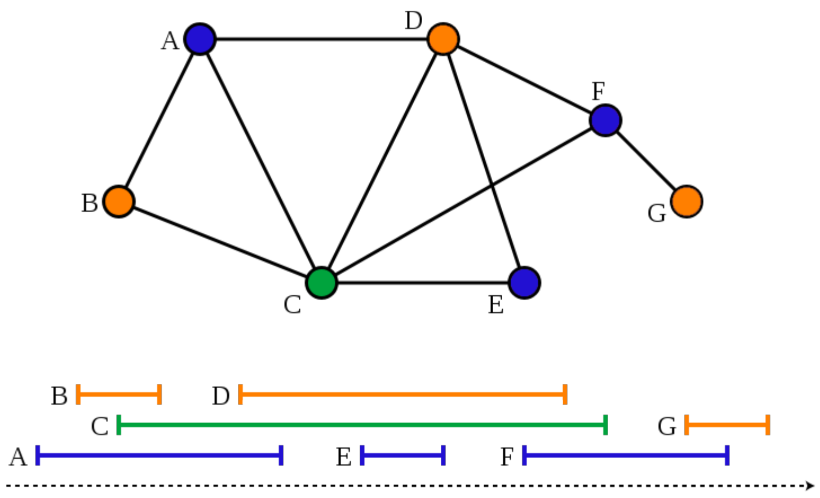
IMPLEMENTING PARALLEL FIRST FIT GRAPH COLORING IN JAVA

CSE 6490A Winter 2011

Loutfouz Zaman

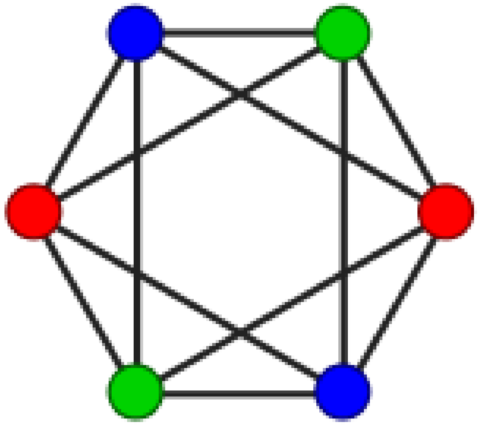


Overview

- Graph coloring revisited
- Sequential FF
- Parallel FF
- Generalized Parallel FF
- CSP example
- Impementation of Generalized Parallel FF explained
- Evaluation
  - Performance
  - Correctness

Vertex coloring

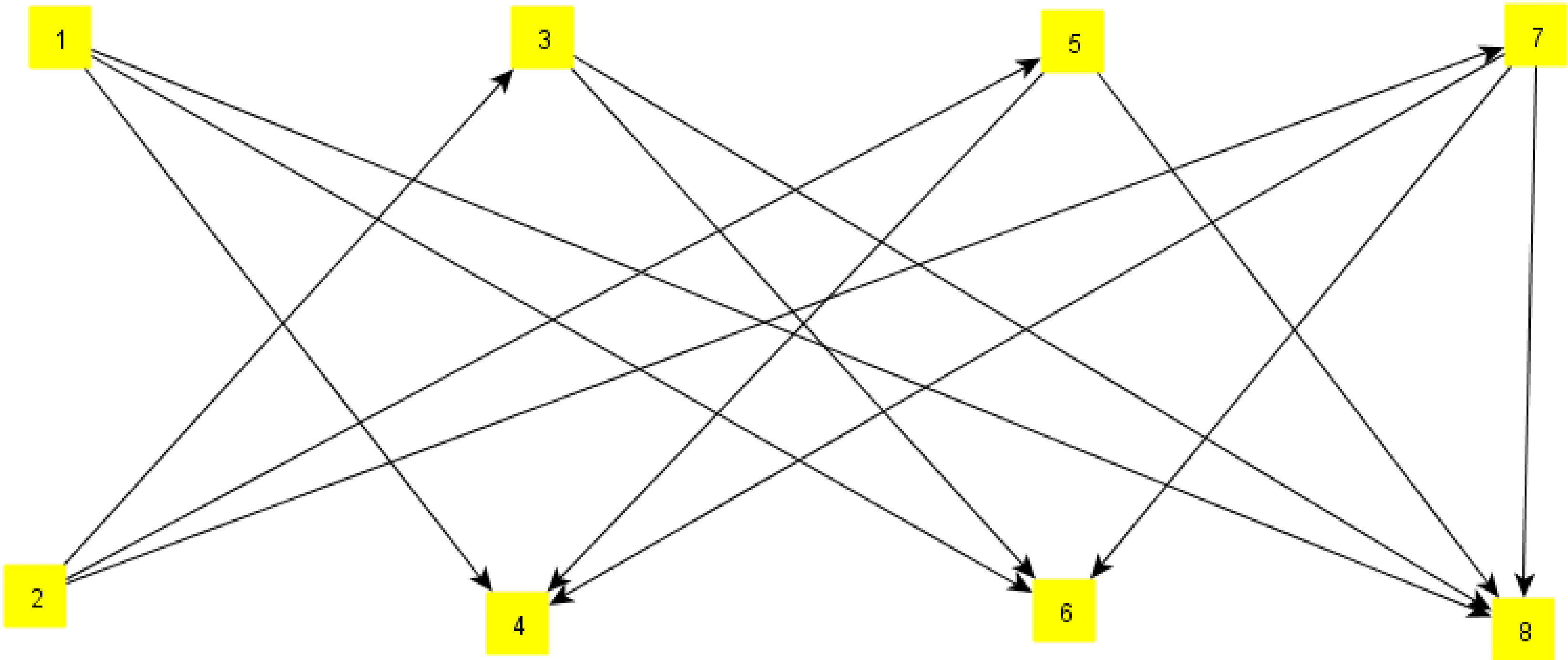
- Assignment of *"colors"* to vertices in a so that no two adjacent vertices share the same color
- First-Fit is the simplest algorithm
  - works by assigning the smallest possible integer as color to the current vertex of the graph



Sequential FF

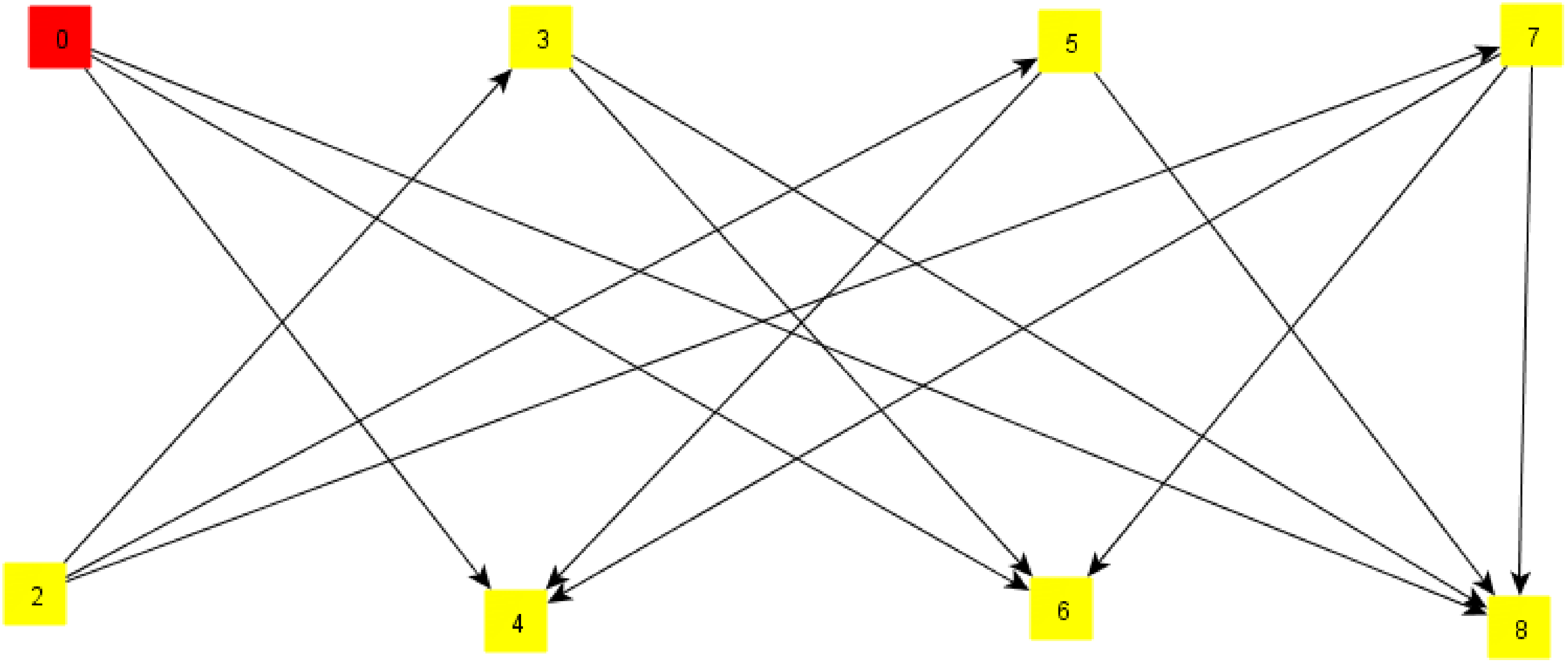
- Umland (1998) demonstrates a 2-step sequential FF algorithm:
  - **(1)** $\boldsymbol{Build(L_i, v_j)}$: Determine a list L_i of all possible colors for v_i, i.e. exclude colors already used by vertices $v_j, j < i$ adjacent to v_i
    - L_i -- a boolean array (possibility list of v_i) with property:
      - $L_i[k] = false \leftrightarrow \exists v_j$ such that $j < i, (v_i, v_j) \in E$ and $f(v_j) = k$
  - **(2)** $\boldsymbol{Color(L_i, v_i)}$: Determine the smallest of all possible colors for v_i, i.e. look for the smallest entry in L_i where $L_i[k] = true$ and assign color k to v_i

Sequential FF E.g. Step 0



Sequential FF E.g. Step 1

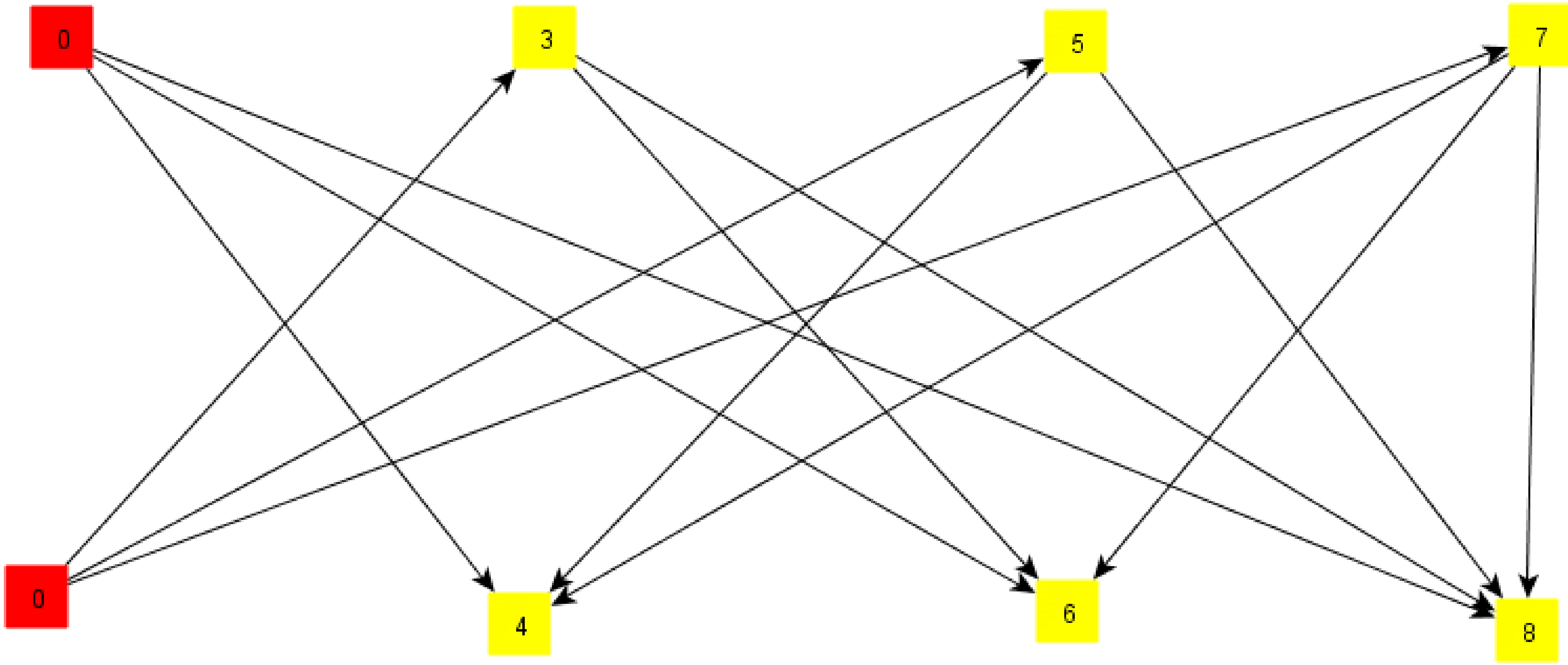
$L_1 = \{t, t, t, t\}$,k=0



$L_4 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$

Sequential FF E.g. Step 2

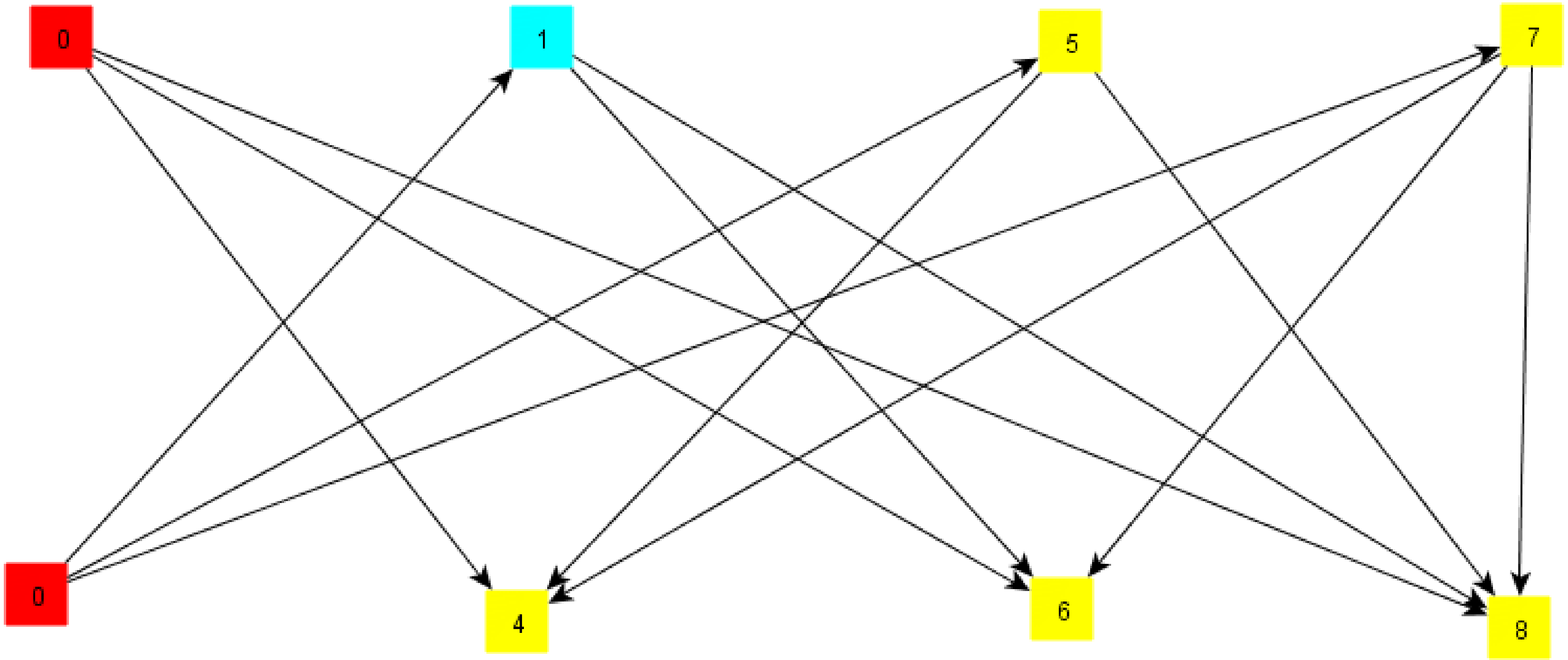
$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}$ $L_5 = \{f, t, t, t\}$ $L_7 = \{f, t, t, t\}$



$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$

Sequential FF E.g. Step 3

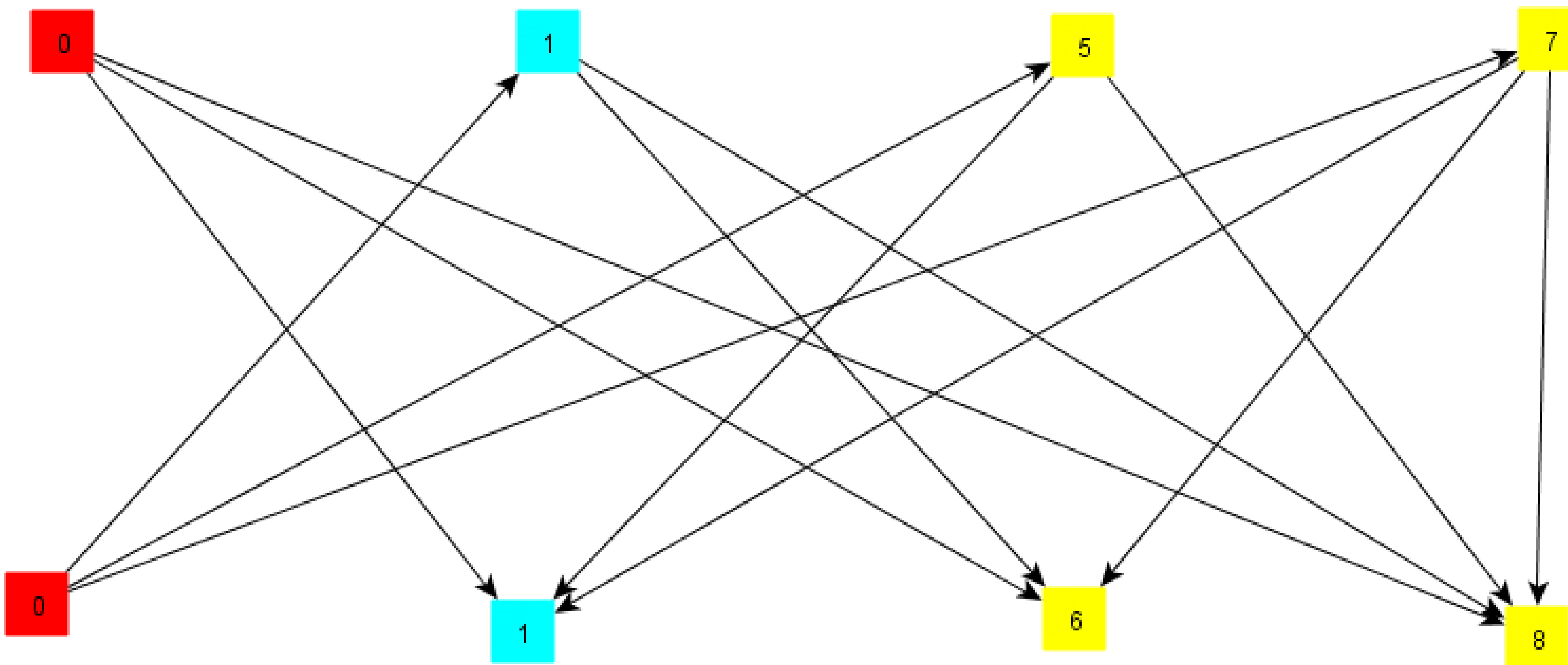
$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k = 1$ $L_5 = \{f, t, t, t\}$ $L_7 = \{f, t, t, t\}$



$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}$ $L_6 = \{f, f, t, t\}$ $L_6 = \{f, f, t, t\}$

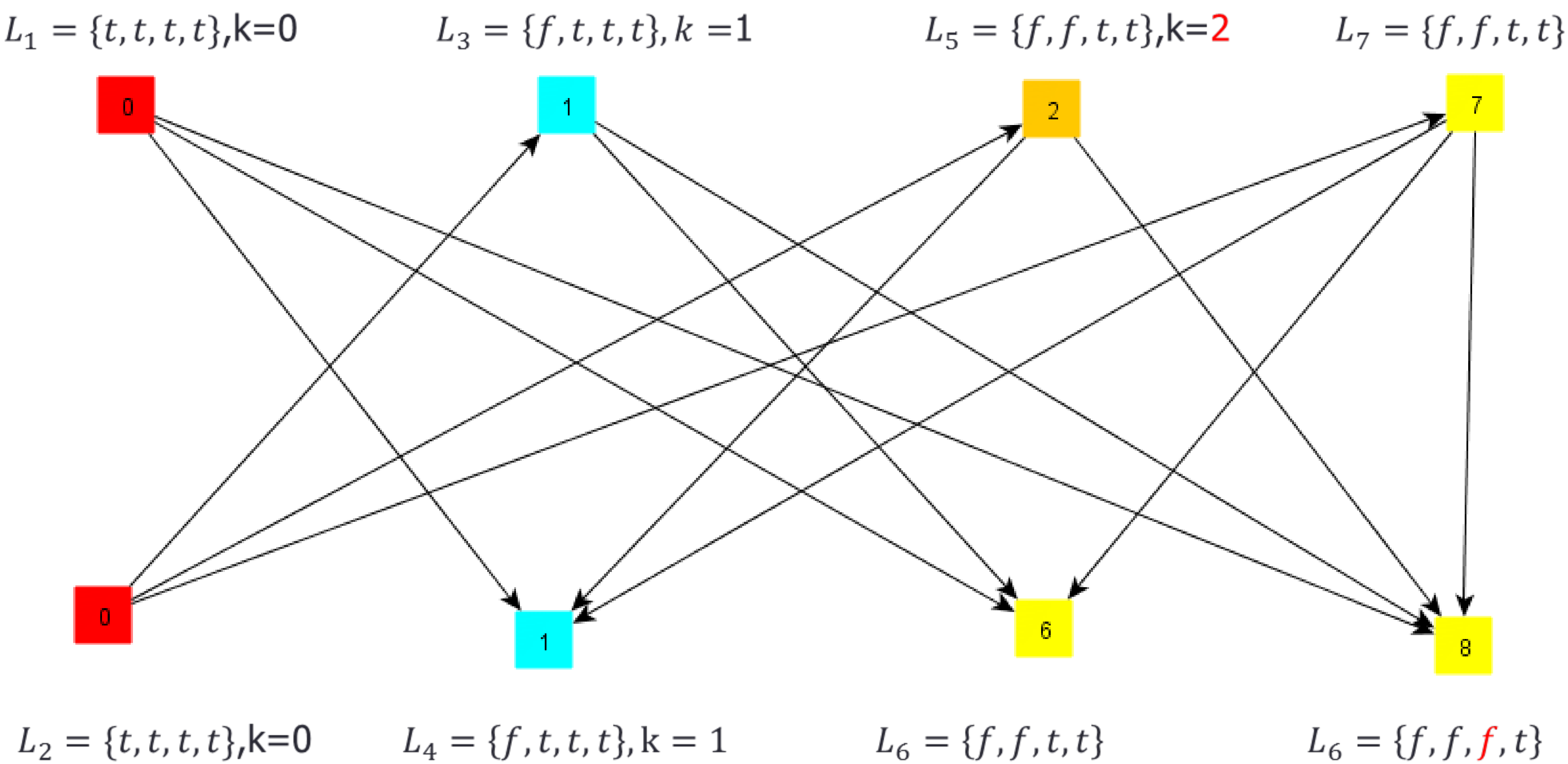
Sequential FF E.g. Step 4

$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k$ =1 $L_5 = \{f, f, t, t\}$ $L_7 = \{f, f, t, t\}$



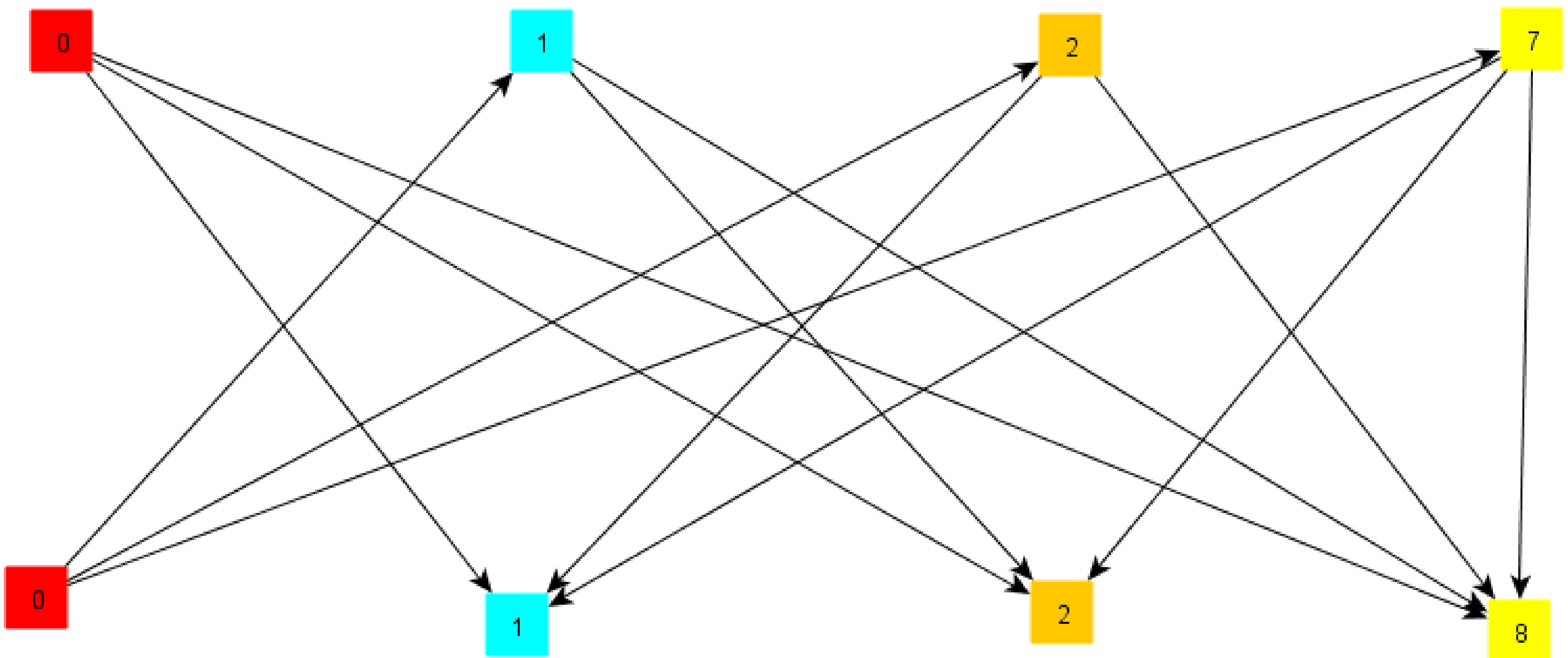
$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}, \mathrm{k} = 1$ $L_6 = \{f, f, t, t\}$ $L_6 = \{f, f, t, t\}$

Sequential FF E.g. Step 5



Sequential FF E.g. Step 6

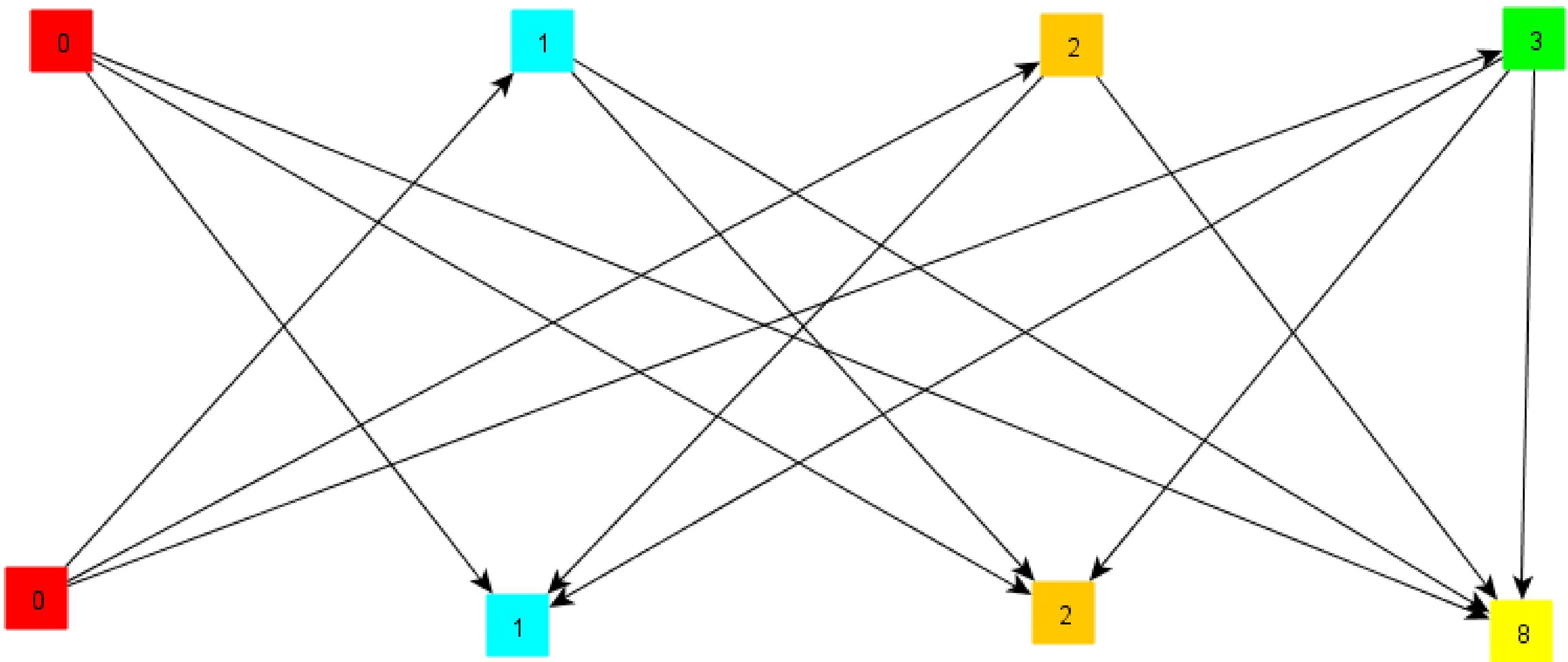
$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k$ =1 $L_5 = \{f, f, t, t\}$,k=2 $L_7 = \{f, f, f, t\}$



$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}, \mathrm{k} = 1$ $L_6 = \{f, f, t, t\}$,k=2 $L_6 = \{f, f, f, t\}$

Sequential FF E.g. Step 7

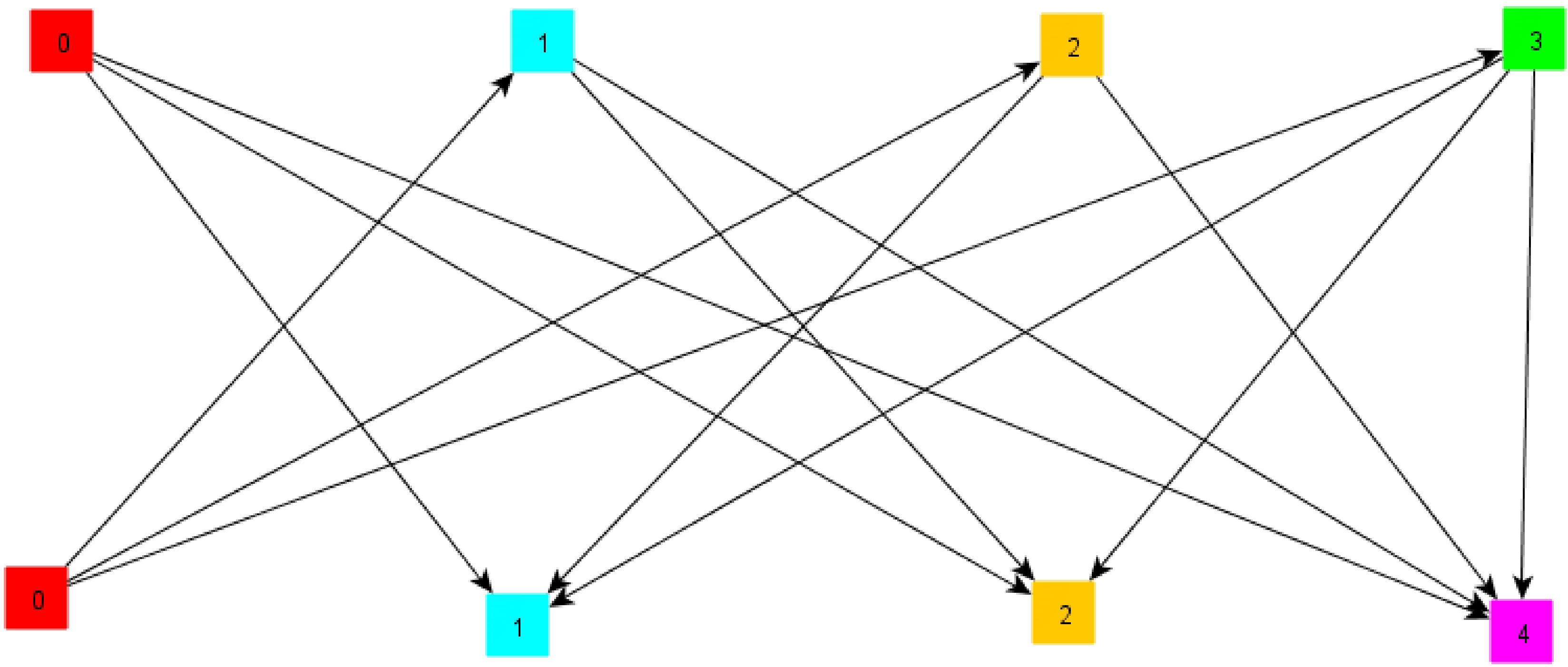
$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k$ =1 $L_5 = \{f, f, t, t\}$,k=2 $L_7 = \{f, f, f, t\}, k = 3$



$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}, \mathrm{k} = 1$ $L_6 = \{f, f, t, t\}$,k=2 $L_6 = \{f, f, f, f\}$

Sequential FF E.g. Step 8

$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k$ =1 $L_5 = \{f, f, t, t\}$,k=2 $L_7 = \{f, f, f, t\}, k = 3$



$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}$, k = 1 $L_6 = \{f, f, t, t\}$,k=2 $L_6 = \{f, f, f, f\}$,k=4

Parallel FF (Vertex Based)

Step	**Processor$_1$**	**Processor$_2$**	**Processor$_3$**	**Processor$_4$**	**Processor$_5$**
1.	Color(L_1, v_1)				
2.	Build(L_2, v_1)				
3.	Build(L_3, v_1)	Color(L_2, v_2)			
4.	Build(L_4, v_1)	Build(L_3, v_2)			
5.	Build(L_5, v_1)	Build(L_4, v_2)	Color(L_3, v_3)		
6.		Build(L_5, v_2)	Build(L_4, v_3)		
7.			Build(L_5, v_3)	Color(L_4, v_4)	
8.				Build(L_5, v_4)	
9.					Color(L_5, v_5)

Figure 2: Parallel first fit with 5 vertices and 5 processors.

Disadvantage:
requires
$n_{vertices} = n_{processors}$

Generalized Parallel FF (Subgraph Based)

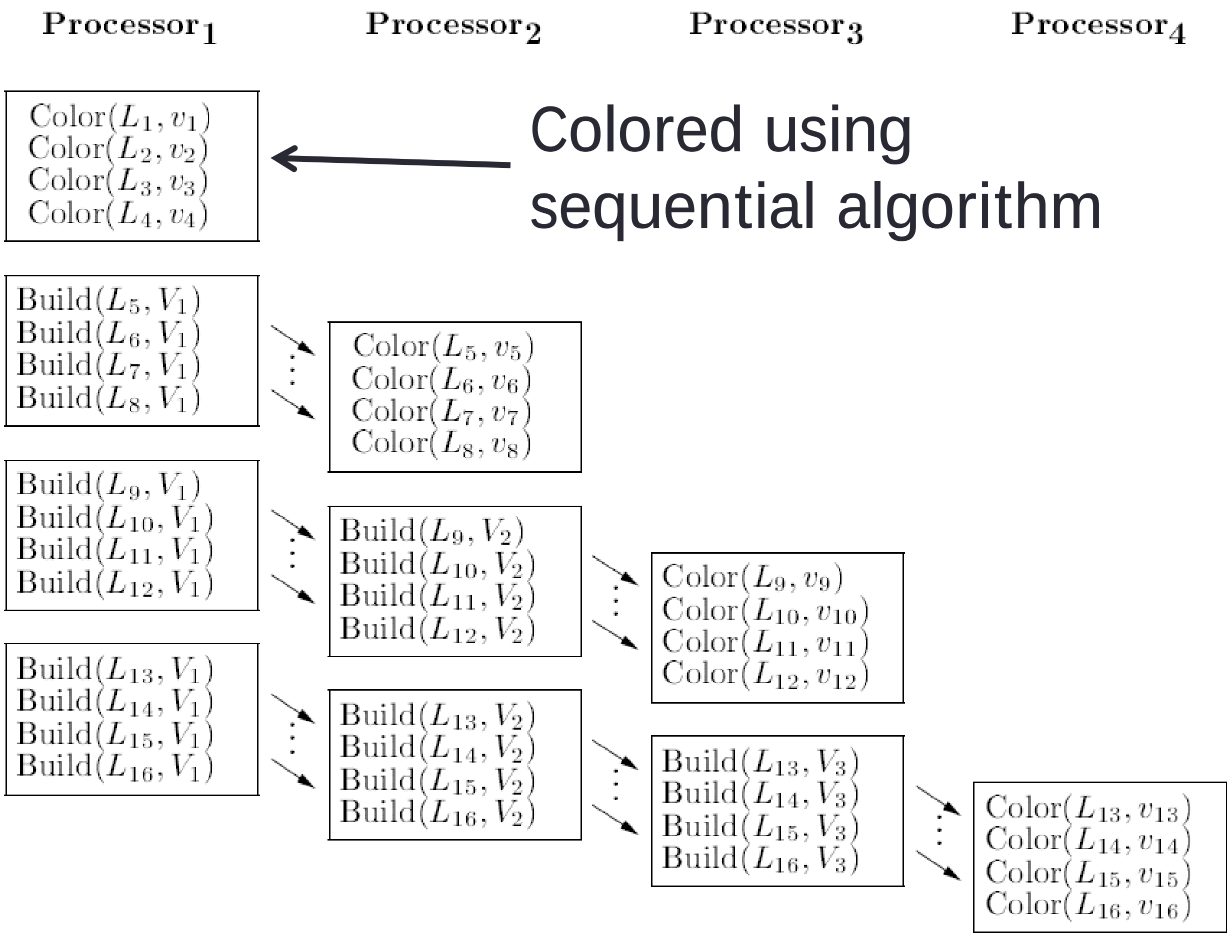


Figure 3: Generalized parallel first fit (16 vertices, 4 processors).

CSP explained using a simple example

```
class CSPDemo
{
        public static void main(String[] args)
        {
                One2OneChannel chan = Channel.one2one();

                  new Parallel
                  (
                        new CSProcess[]
                        {
                          new SendEvenIntsProcess (chan.out()),
                          new ReadEvenIntsProcess (chan.in())
                        }
                  ).run ();
        }
}
```

Writer

```
class SendEvenIntsProcess implements CSProcess
{
    private ChannelOutput out;

    public SendEvenIntsProcess(ChannelOutput out)
    {
      this.out = out;
    }

    public void run()
    {
      for (int i = 2; i <= 100; i = i + 2)
      {
        out.write (new Integer (i));
      }

    }

}
```

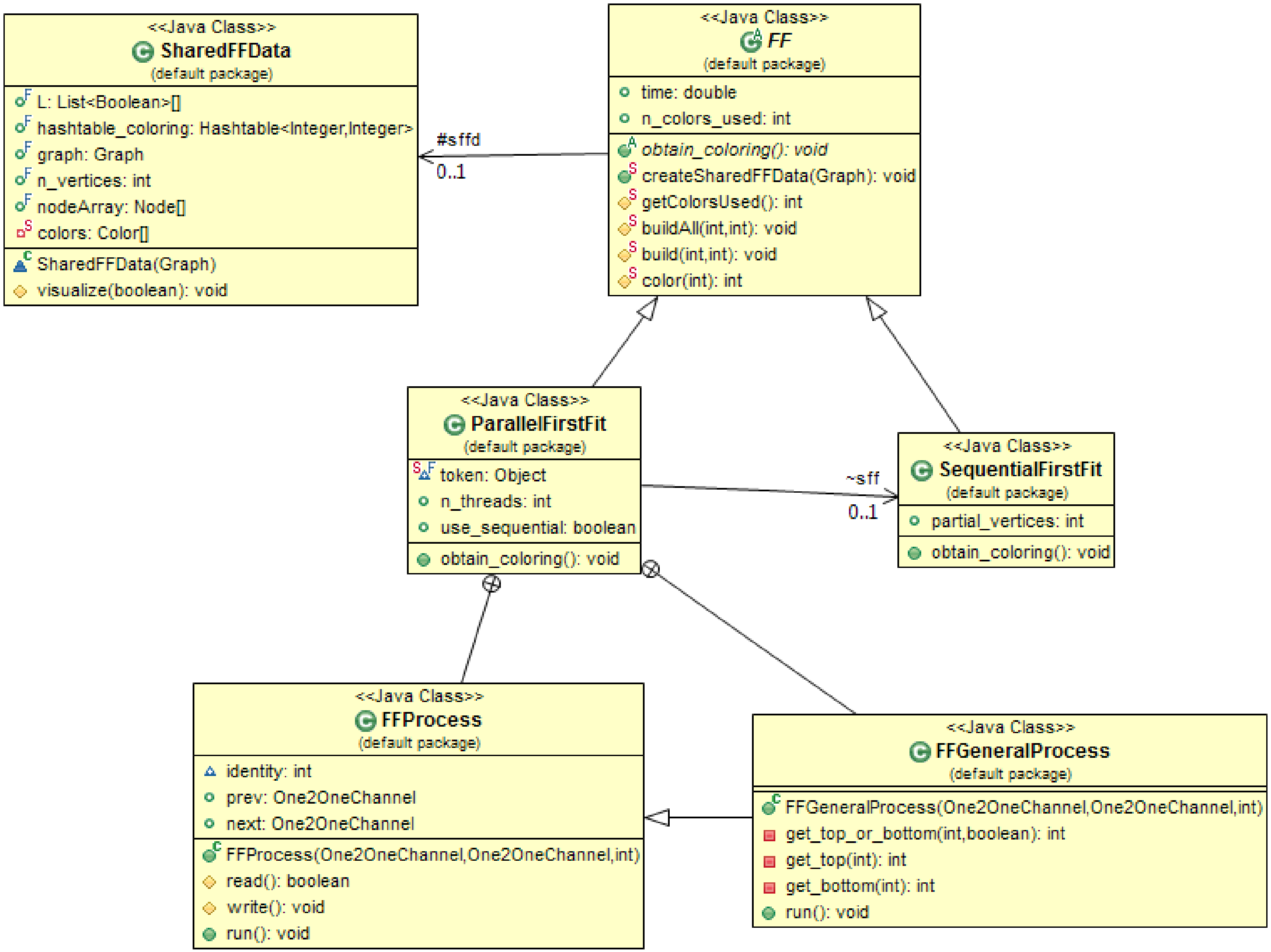
Reader

```
class ReadEvenIntsProcess implements CSProcess
{
      private ChannelInput in;
      public ReadEvenIntsProcess(ChannelInput in)
      {
         this.in = in;
      }

      public void run()
      {
         Integer d = 0;
         while (d < 100)
         {
            d = (Integer)in.read();
            System.out.println("Read: " + d.intValue());
         }
      }
}
```

Output

- Read: 2
- Read: 4
- Read: 6
- …
- Read: 100

Class Diagram for FF Implementation



Creating Parallel CSP Process

```
public void obtain_coloring()
{
        One2OneChannel prev = null, next = null;
        //create channel between this process and previous process
        //create channel between this process and next process
        //first process doesn't have channel to previous process
        for (int i = 0; i < n_threads; i++)
        {
                next = Channel.one2one();
                if (i == n_threads - 1)
                        next = null; //last process doesn't have channel to next proccess
                csprocesses[i] = new FFGeneralProcess(prev,next,i);
                prev = next;
        }
        //construct a parallel
        Parallel parallel = new Parallel(csprocesses);
        //run parallel
        parallel.run();

}
```

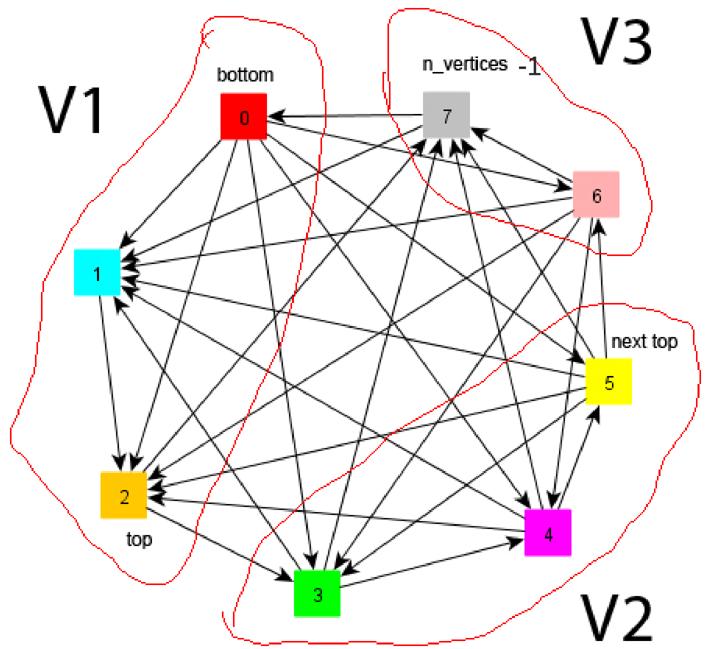
Synchronization using CSP token passing

```
protected boolean read()
{
        try
        {
                prev.in().read();
        }
        catch (NullPointerException e)
        {
                //first process will not have a prev channel,
                //therefore the process shouldn't get stuck
                return true; //returns true exception
        }
        return false;
}

protected void write()
{
        try
        {
                next.out().write(token);
        }
        catch (NullPointerException e)
        {
                //last process will not have a next channel,
                //therefore the process shouldn't get stuck
        }
}
```

If N doesn't divide n,
The subgraphs have equal number of vertices except the last one which has less

```
public void run()
{
        //obtain subgraph vertices
        int bottom = get_bottom(identity);
        int top = get_top(identity);

        if (identity == 0)
        {
                ParallelFirstFit.sff.partial_vertices = top;
                ParallelFirstFit.sff.obtain_coloring();
        }
        else
        {
                for (int i = bottom; i < top; i++)
                {
                        if (read()) //if it throws an exception
                        {
                                //this is first time, so we give the first color
                                sffd.hashtable_coloring.put(i, i);
                        }
                        else
                        {
                                color(i);
                        }
                }
        }

        //last processor doesn't do any building, so quit here
        if (identity == n_threads - 1)
        {
                return;
        }

        int next_top =  get_top(identity + 1);
        top = get_bottom(identity+1)
        for (int j = top; j < next_top; j++)
        {
                Node nextNode = null;

                 nextNode = sffd.graph.getNodeArray()[j];

                //subgraph iteration
                read();
                for (int i = bottom; i < j; i++)
                {
                        Node node = sffd.nodeArray[i];
                        //if there is an edge, you build
                        if (nextNode.getEdge(node) != null)
                        {
                                int color = (Integer) sffd.hashtable_coloring.get(i);
                                build(color, j);
                        }
                }
                write();
        }

        for (int j = next_top; j < sffd.n_vertices; j++)
        {
                Node nextNode = sffd.graph.getNodeArray()[j];

                //subgraph iteration
                read();
                for (int i = bottom; i < top; i++)
                {
                        Node node = sffd.nodeArray[i];
                        //if there is an edge, you build
                        if (nextNode.getEdge(node) != null)
                        {
                                int color = (Integer) sffd.hashtable_coloring.get(i);
                                build(color, j);
                        }
                }
                write();
        }

}
```

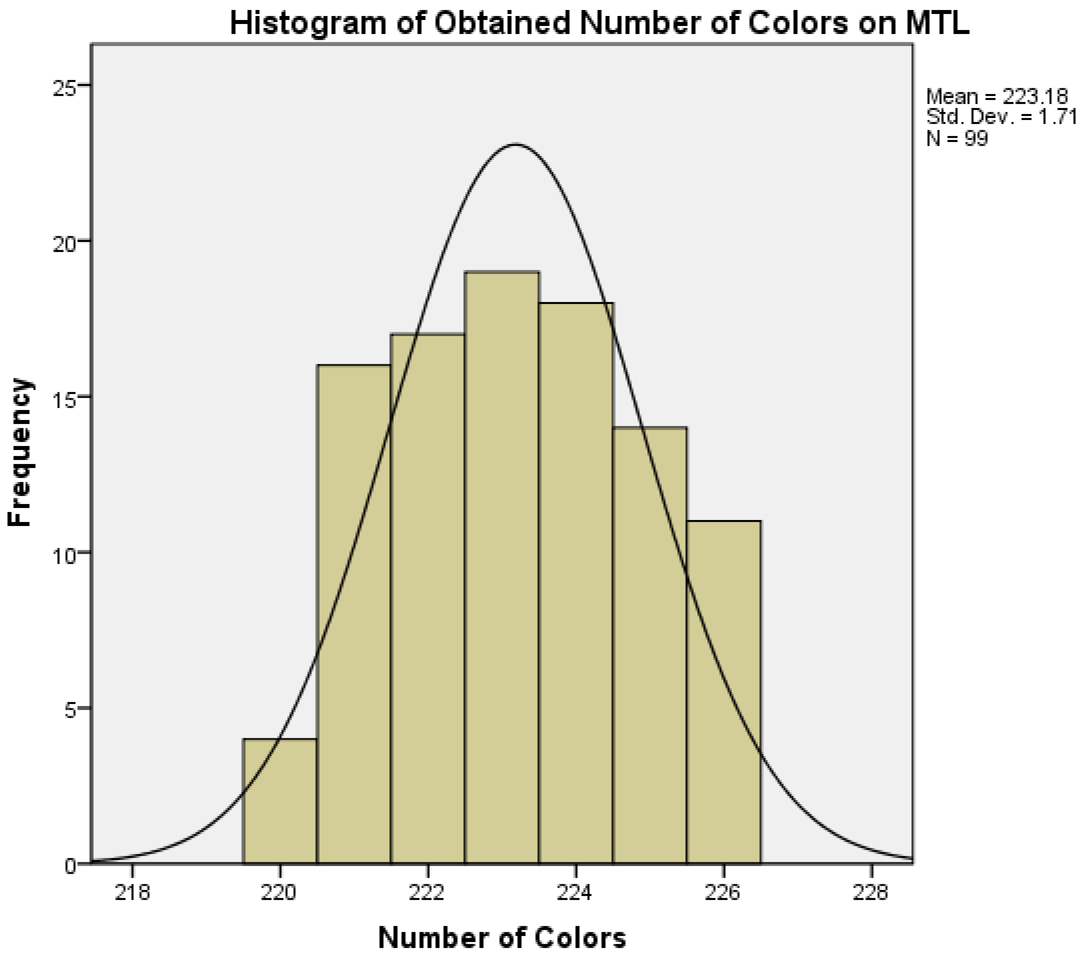
Outside of csp process, uses synch object

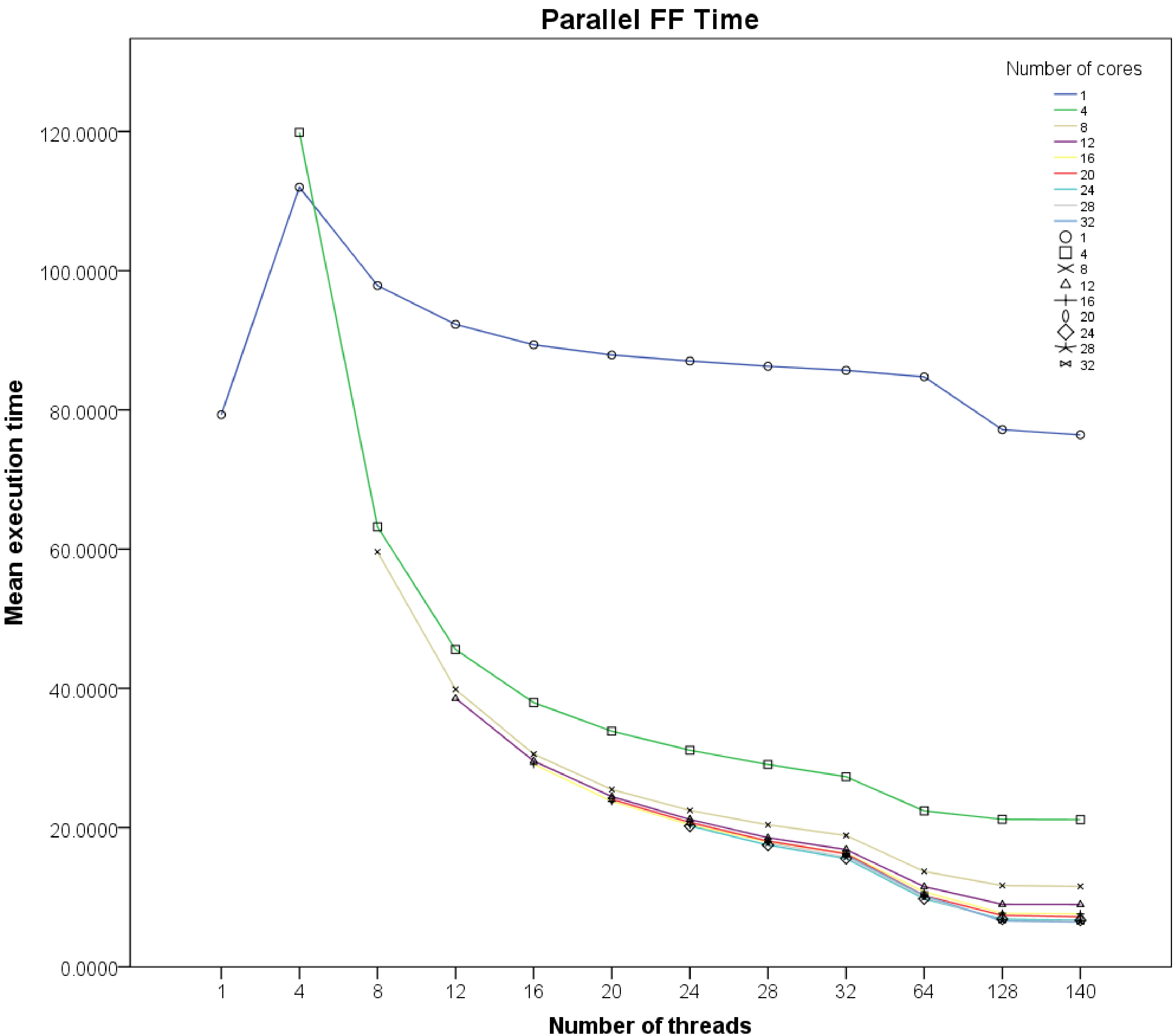


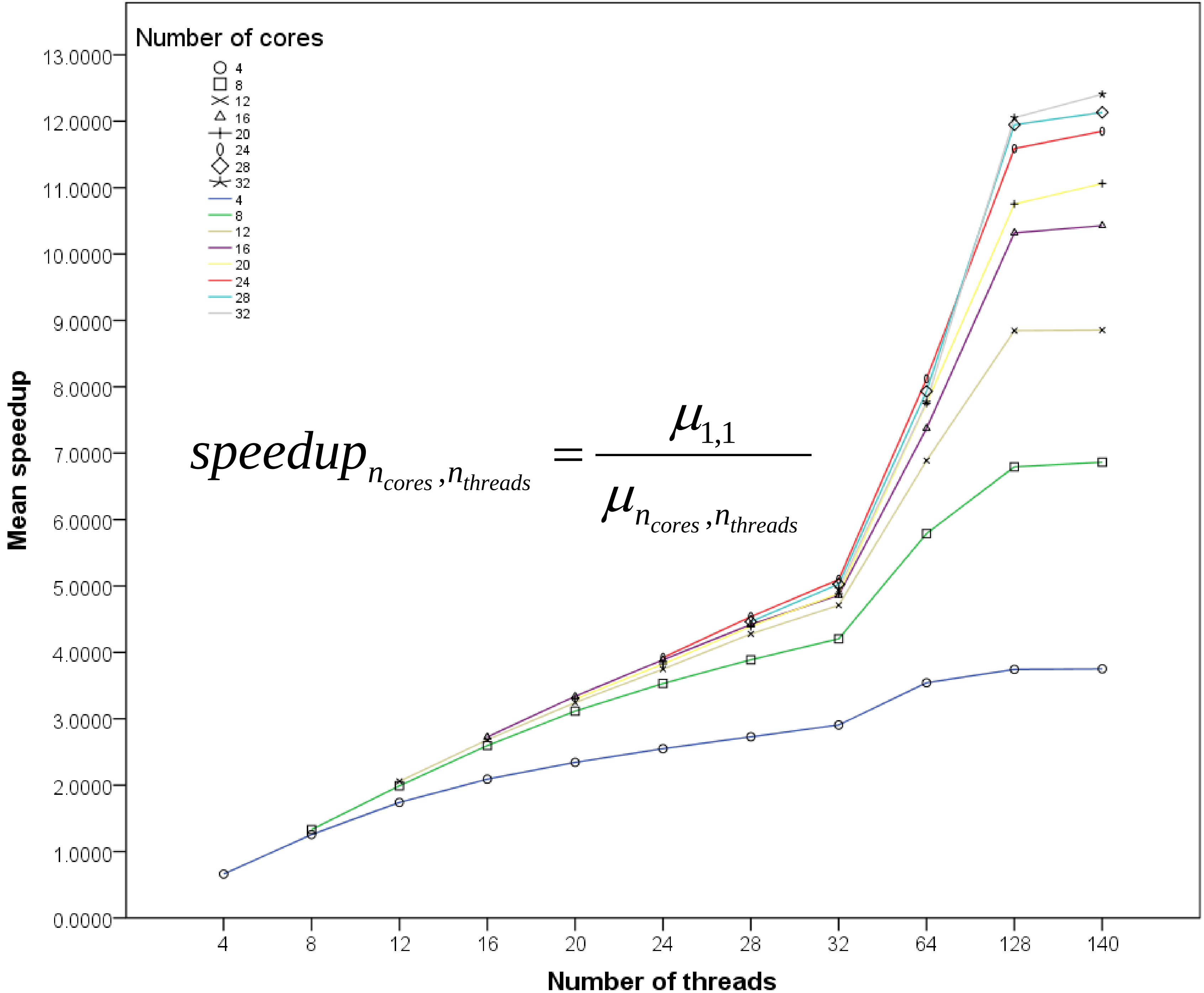
Evaluation

- Time to generate graph was not counted
  - Pre-generated for all trials
  - 2,000 vertices and 999,001 edges
- On MTL
  - For 1,4,8,12,16,20,24,28,32 cores
    - For 1,4,8,12,16,20,24,28,32,64,128,140 threads
      - For 12 iterations
- Took approx. 10 hours
- Iteration=0 not reported

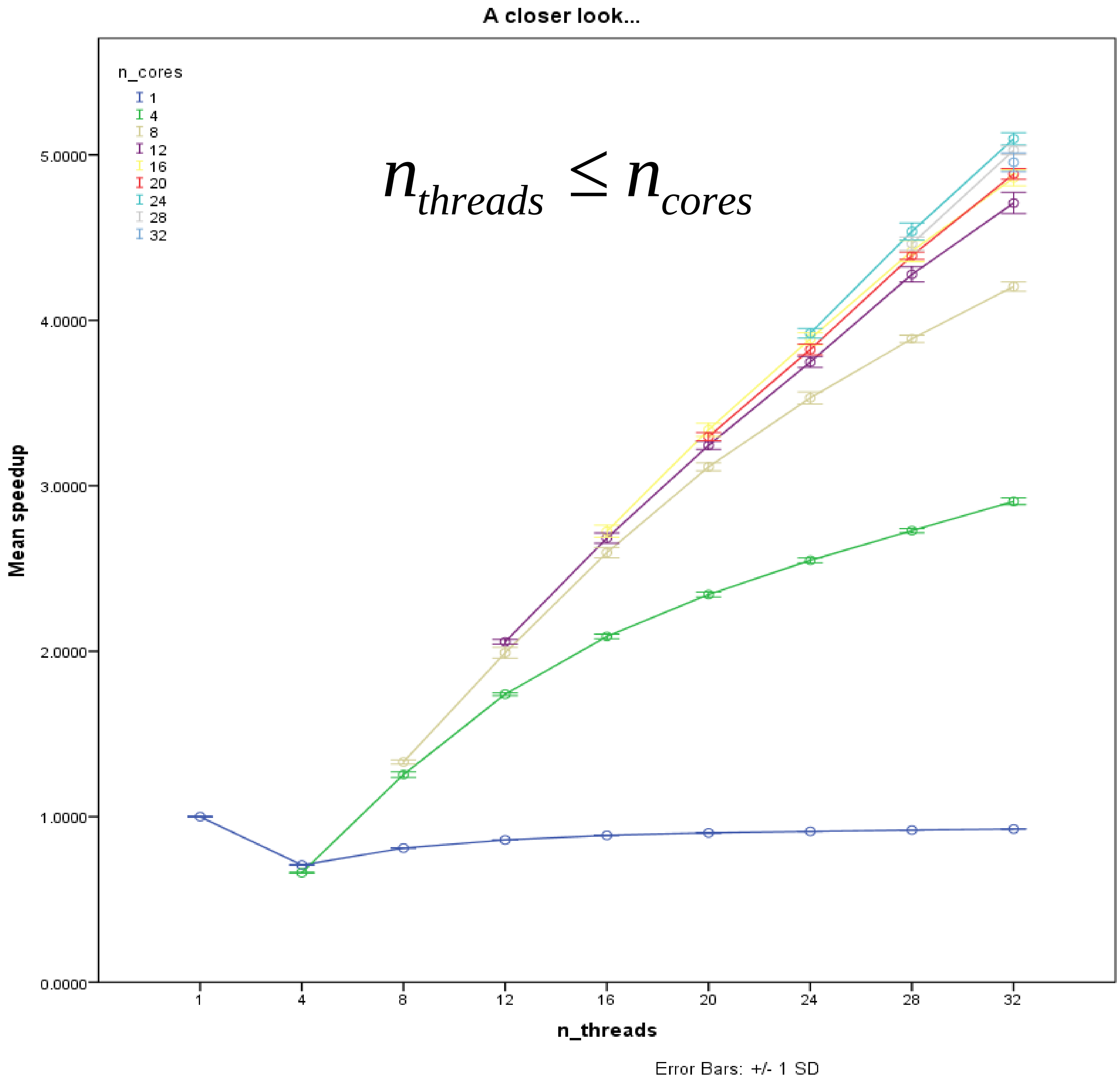
Obtained colors

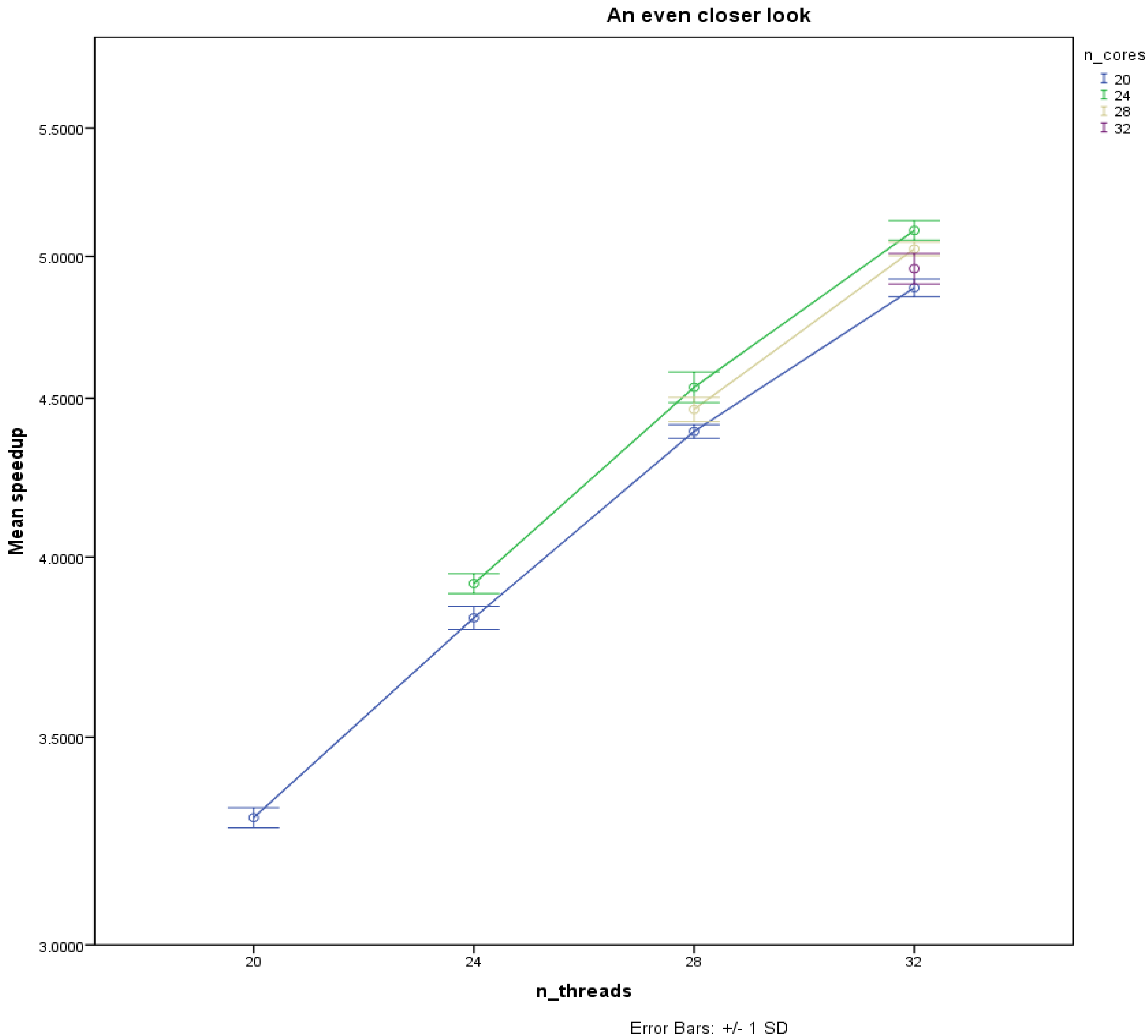



Parallel FF Time
Number of cores
1
4
8
12
16
20
24
28
32
Mean execution time
120.0000
100.0000
80.0000
60.0000
40.0000
20.0000
0.0000
1
4
8
12
16
20
24
28
32
64
128
140
Number of threads

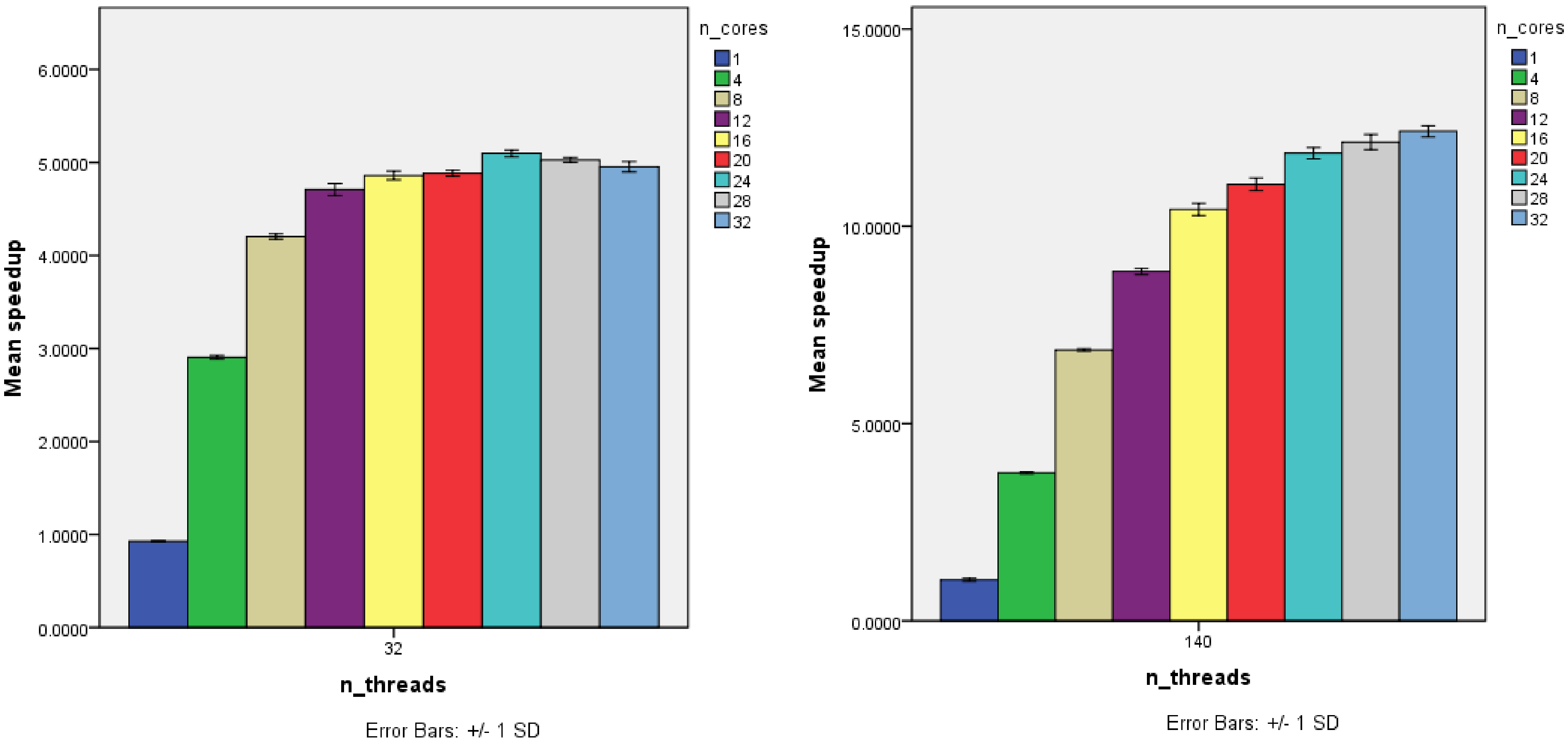


$$speedup_{n_{cores},n_{threads}} = \frac{\mu_{1,1}}{\mu_{n_{cores},n_{threads}}}$$


A closer look...
n_cores
1
4
8
12
16
20
24
28
32
$n_{threads} \leq n_{cores}$
Mean speedup
0.0000
1.0000
2.0000
3.0000
4.0000
5.0000
1
4
8
12
16
20
24
28
32
n_threads
Error Bars: +/- 1 SD


An even closer look
n_cores
20
24
28
32
5.5000
5.0000
4.5000
4.0000
3.5000
3.0000
Mean speedup
20
24
28
32
n_threads
Error Bars: +/- 1 SD

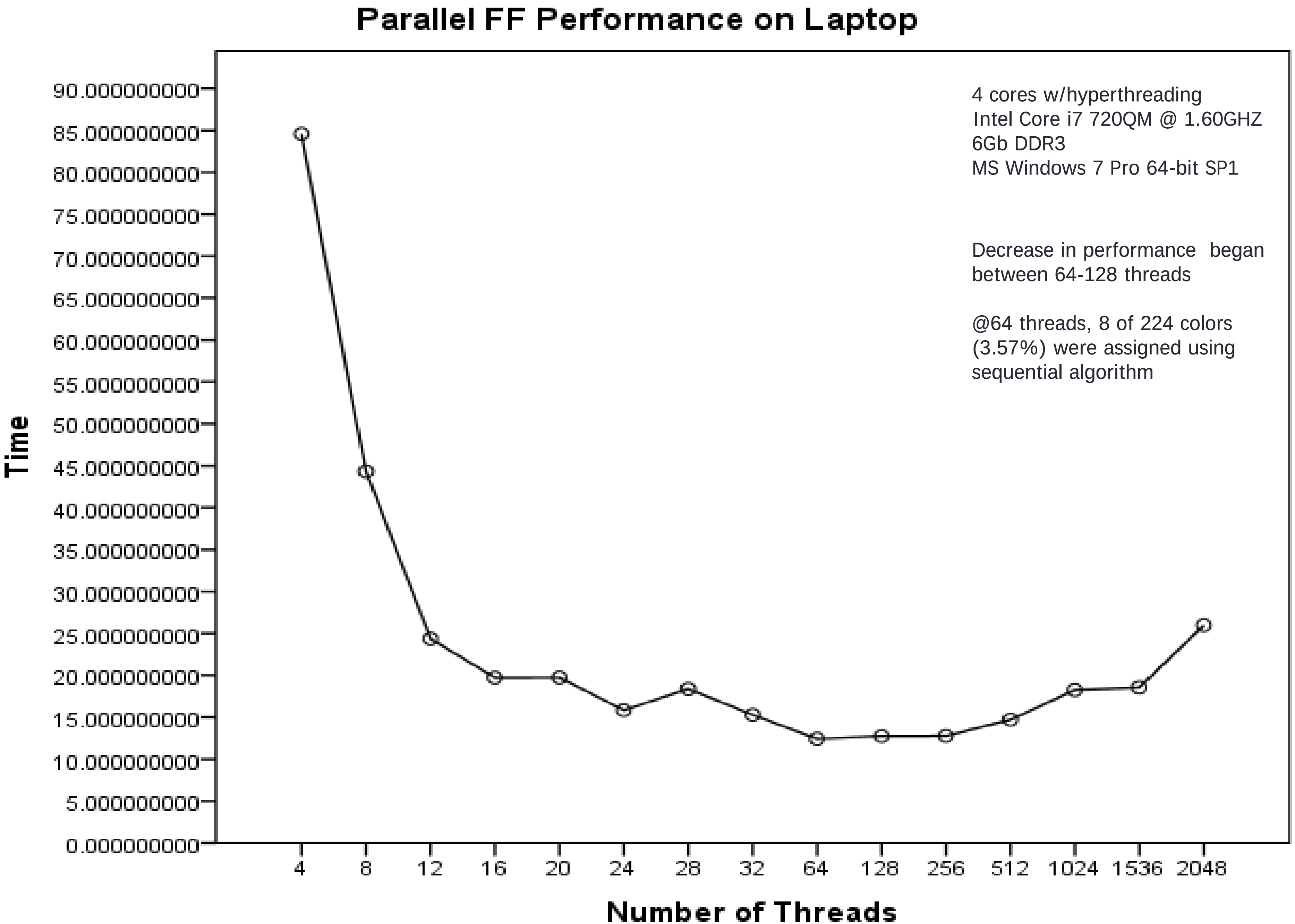
Speedup for $n_{threads}$=32;140

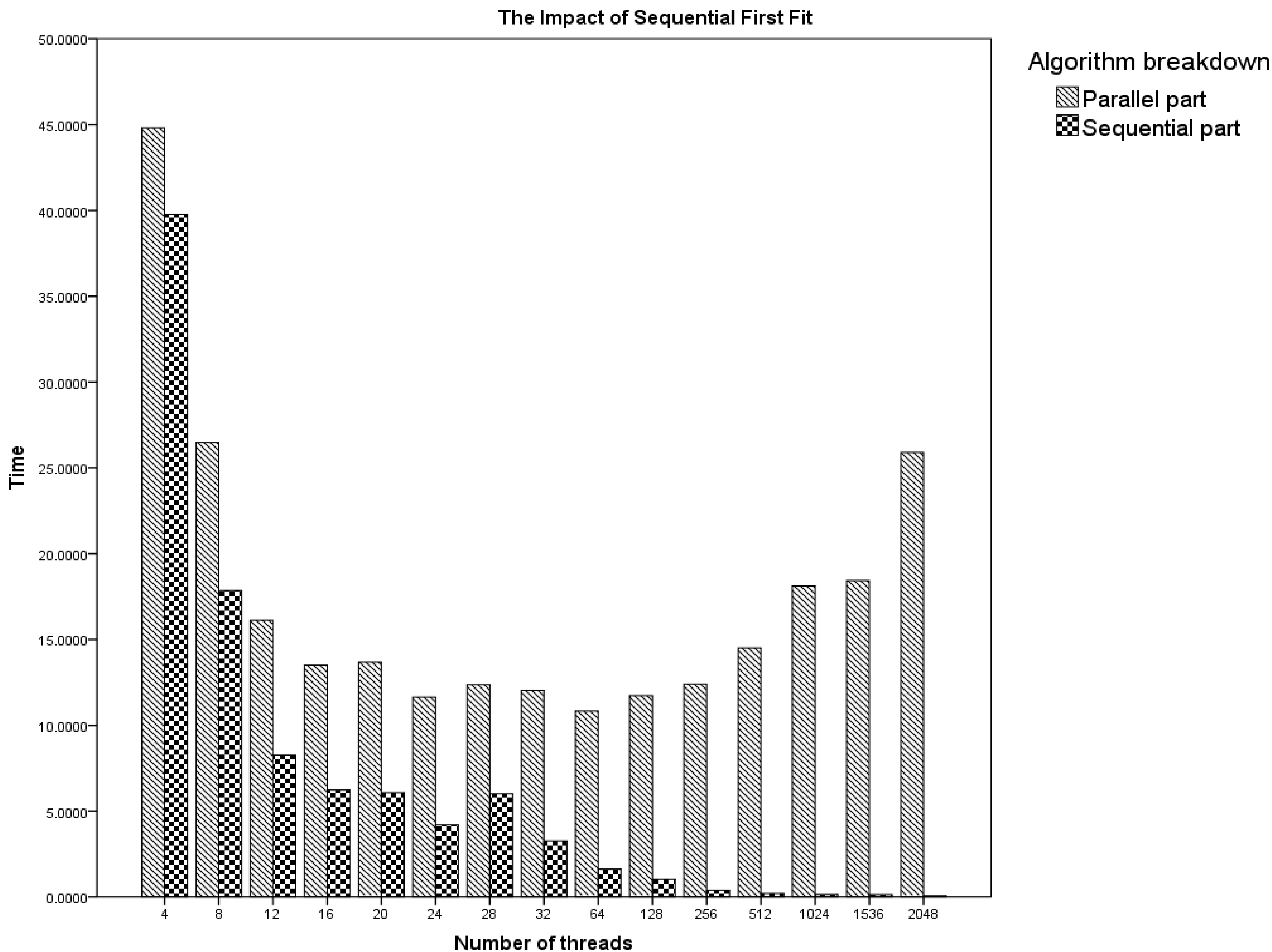


Peak performance

- Was not reached due to 140 threads limitation on MTL
- Single iteration on Laptop to investigate peak performance
  - 4 cores with hyperthreading
  - Intel Core i7 720QM @ 1.60GHZ
  - 6Gb DDR3
  - MS Windows 7 Pro 64-bit SP1
- Investigated

$$n_{threads} = 2^2, 2^3, \ldots, 1536, 2^{11}$$


Parallel FF Performance on Laptop
4 cores w/hyperthreading
Intel Core i7 720QM @ 1.60GHZ
6Gb DDR3
MS Windows 7 Pro 64-bit SP1
Decrease in performance began between 64-128 threads
@64 threads, 8 of 224 colors (3.57%) were assigned using sequential algorithm
Time
90.000000000
85.000000000
80.000000000
75.000000000
70.000000000
65.000000000
60.000000000
55.000000000
50.000000000
45.000000000
40.000000000
35.000000000
30.000000000
25.000000000
20.000000000
15.000000000
10.000000000
5.000000000
0.000000000
4
8
12
16
20
24
28
32
64
128
256
512
1024
1536
2048
Number of Threads


The Impact of Sequential First Fit
Time
50.0000
45.0000
40.0000
35.0000
30.0000
25.0000
20.0000
15.0000
10.0000
5.0000
0.0000
4
8
12
16
20
24
28
32
64
128
256
512
1024
1536
2048
Number of threads
Algorithm breakdown
Parallel part
Sequential part

Test for correctness

- Test cases created using viz tool introduced in assignment 1
  - Helped greatly!
- All subgraph partitioning scenarios were tested too
  - Graphs were picked to test all possible subgraph partitioning scenarios, E.g. 8 node graph:
    - $P_1 \rightarrow$ nodes 1-1, $P_2 \rightarrow$ nodes 2-2,…, $P_8 \rightarrow$ nodes 8-8
    - $P_1 \rightarrow$ nodes 1-2, $P_2 \rightarrow$ nodes 3-4,…, $P_4 \rightarrow$ nodes 7-8
    - $P_1 \rightarrow$ nodes 1-3, $P_2 \rightarrow$ nodes 4-6, $P_3 \rightarrow$ nodes 7-8, etc…
- Print statements and sorting the output alphabetically:
  - System.out.println(“CSProccess id:”+identity+”,some test stuff”);
  - I know it’s not the best way!
  - Will try pathfinder in assignment 3

Conclusion & Future Work

- Based on observation algorithm performs better when

$$n_{threads} > n_{cores}$$

- Consistent (to some extent) with Umland's (1998) findings
- Modeling
  - optimal $n_{threads}$ should be predictable using
    - n_{cores}, $n_{vertices}$, n_{edges}, and possibly other variables

Questions?

- References
  - Thomas Umland. Parallel graph coloring using JAVA. In *Architectures, Languages and Patterns for Parallel and Distributed Applications, pages 211–218. IOS Press, 1998.*