6.895 Theory of Parallel Systems Project Paper

Conversion of NSP DAGs to SP DAGs

Sajindra Jayasena and Sharad Ganesh

Abstract

Random multithreaded NSP task graphs cannot be modelled in SP-based parallel programming models. In this report we present a new algorithm for converting NSP DAGs to SP DAGs. Our analysis establishes an upper bound of a factor of 2, on the increase in critical path length(W_∞). However, the total work (W_1) remains the same. We also evaluate an implementation of our algorithm and perform subsequent analysis with real-world models of task graphs and random task graphs which provide encouraging results as compared to previous research. Specifically, we prove the correctness and analyze computational complexity of our algorithm followed by a heuristic based variant for further performance enhancements.

1 Introduction

A parallel multithreaded computation, represented as a non-series parallel DAG, presents inherent difficulties in performing cost estimations, analysis, scheduling and structured programming[1]. Also, most of the current multithreaded programming models like *cilk*[2] are based on series-parallel DAG modeling of the computations. However, most of the real world problems that are modeled for parallel system solutions are non-series-parallel DAGs. This exposes a lacuna which necessitates the development of a mechanism for an efficient translation technique from the NSP model to an SP model. The SP model enables the analysis and execution of such programs in the underlying multithreaded environment while preserving the original semantics of the computation. However, it should be noted that we mainly intend to address the conversion of NSP DAG representing the task nodes of a computation to an SP DAG. Further representation is required, if it is to be modeled in a particular multi-threaded computational model like *cilk*[2][3].

The remainder of the report is organized as follows. Section 2 introduces some terms and concepts used by the algorithm. Section 3 presents our approach and strategy, in addition to the proof of concepts used by our algorithm. Section 4 presents the high-level algorithm. Section 5 presents the proof of correctness for the algorithm, while Section 6 discusses the complexity analysis. Section 7 presents a qualitative assessment of the algorithm. Section 8 outlines the empirical analysis followed by some concluding remarks.

2 Some terms and concepts:

Here we outline some basic definitions that are used in the subsequent sections of the report.

Definition 1 *Directed acyclic graph (DAG)*
A graph G is a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices $E = (u, v)|u, v \in V$. *The graph does not allow self-loops, adjacency is irreflexive, i.e.* $E = (u, v)|(u, v \in V) \bigwedge (u \neq v)$ *and has no path which starts and ends at the same vertex.*

A DAG describes the structure of a parallel program. The nodes of the DAG represent tasks that the processors must complete, and the edges represent the dependencies between the tasks. Thus, if there is an edge $(u, v) \in E$, *then v cannot be executed until after u completes. In this case we way that u is a parent of v.*

Definition 2 *Series parallel DAG(SP DAG)*
A series-parallel DAG G' = (V, E) is a directed acyclic graph with two distinguished vertices, a source 's' and a sink 't'. A family of series parallel graphs are described using the following grammar:
A series parallel DAG G'(V, E) is one of the following:

A single edge extending from s to t, that is, $V = \{s, t\}$ *and* $E = \{(s, t)\}$.

Two series parallel graphs G_1 *and* G_2 composed in parallel. *The sources* s_1 *and* s_2 *of* G_1 *and* G_2 *are merged into a single source s, and the sinks* t_1 *and* t_2 *are merged into a single sink t.*

Two series parallel graphs composed in series . *The sink* t_1 *of* G_1 *and the source* s_2 *of* G_2 *are merged into a single node.*

Definition 3 *Non-series parallel DAG(NSP DAG)*
A graph G that cannot be reduced to a representation addressed by the SP DAG.[Definition 2].

Definition 4 *Critical path (*W_∞ *)*
The number of nodes in the longest chain in the DAG G.

Definition 5 *Total Work (*W_1 *)*
The total number of nodes in the DAG G.

Definition 6 *Level of a vertex in a DAG (LEVEL(v))*
The longest path length from the root to the vertex v.

Definition 7 *Predecessor(s) of a vertex (PREDECESSOR(v))*
The vertices that precede v, in a topological ordering of the DAG.

Definition 8 *Successors(s) of a vertex (SUCCESSOR(v))*
The vertices that succeed v, in a topological ordering of the DAG.

Definition 9 *Sync node*
A vertex with indegree ≥ 2.

Definition 10 *Spawn node*
A vertex with outdegree ≥ 2.

Definition 11 *Path*
A path is a sequence of edges that defines a possible way through connected vertices from one vertex to another.

Definition 12 *Spawn Set (SPAWN(v))*
The set of immediate $PREDECESSOR(v)$*, from which* v *was spawned. This identifies the nodes that sync from multiple-independent execution paths.*

Definition 13 *Disjoint vertices*
Two vertices v_i *and* v_j *are disjoint if:*

$$SPAWN(v_i) \cap SPAWN(v_j) = \phi$$

Definition 14 *Disjoint sync node(DSN)*
A node is a disjoint sync node if any of its parents are disjoint vertices.

Definition 15 *Global-Level*
The level of the most-recent disjoint sync node in a topological ordering of the DAG.

Definition 16

C bit of a vertex: Represents the resolved status of a vertex.

R bit of a vertex: Represents node(s) in the NSP DAG, which are the most recently processed vertices in the resultant SP DAG.

D-bit of a supernode: The D bit of a super node S is set when $\exists$ *vertices* $v_i, v_j \in S$*, such that* v_i*,* v_j *are disjoint vertices.*

Definition 17 *Depth(G)*
The level of the leaf vertex v $\in G$.

3 Approach and Strategy

This section addresses the main concepts and strategies that are the cornerstones of our algorithm. Each conceptual idea is justified by its correctness and ability to perform a correct and efficient transformation.

3.1 Breadth first task node visitation (BFT)

Our high level algorithm is based on the breadth first traversal of the task nodes in the NSP DAG. The objective is to identify non-interdependent task nodes that could be run in parallel. At each level, the algorithm resolves the NSP problems (Appendix A), if it exists. There is a subtle difference in the way our algorithm performs the breadth-first traversal of the graph as compared to the normal BFT. In the BFT we follow, a node is enqueued if and only if all its parents have been enqueued and processed in a previous iteration.

The following lemma shows that all nodes considered in an iteration of the breadth first traversal are in the same level.

Lemma 1 *Let* $v_1, v_2, v_3, ...v_m$ *be the vertices considered in a particular iteration of the BFT.*

Then, $\forall$ *i,j where i,j* $\in \{1... \ m\}$*, Level(*v_i*) = Level(*v_j*),* $i \neq j$

Proof A breadth-first traversal of the DAG G is a level-wise traversal, which does not advance to the next level, Level(j) unless and until all the vertices {V} of Level(i) [$i < j$] are visited and colored. By the inherent method of breadth first traversal, the parents of a vertex are discovered and colored before their children are discovered and colored. Hence, all the nodes at a particular BFT are at same level, due to the way in which breadth-first traversal discovers the vertices. □

Further the following Lemma proves that the precedence relation is maintained during each iteration of the BFT.

Lemma 2 *The resolution of a vertex v preserves the precedence relation.*

Proof A vertex $v \in G$ is resolved only if *Parents(v)* have been colored and resolved. A vertex $v \in V$ might have multiple paths $p_1, p_2, ...p_n$. A vertex v is colored and resolved only if the vertex has been reached through the longest path p_i. Thus, without loss of generality, if it has been reached through the longest path, all the other paths to vertex v, would have been traversed [Lemma 1]. Thus, the resolution of a vertex, would never lead to a situation that would violate the precedence. □

Now we would introduce another set of definitions.

Definition 18 *Vertex Coloring*
A vertex v in the NSP DAG G is ***C****-colored if it has been traversed and resolved. Further a vertex is resolved if it is added to the resultant SP DAG G'. The resolved status is represented by a bit known as the C-bit, associated with each vertex of the NSP DAG. If the C-bit of v is set, we call v a colored vertex.*
Further a vertex is ***R****-colored (using a R-bit) if it has already been resolved and does not have any descendants upto the current processing level in G'. The R-coloring propagates down at each iteration to mark the last frontier of processing in G', where all those parents of the newly R-colored vertices, are un-colored(*$R = 0$*).*

This results in another definition.

Definition 19 *Augmented-In-Degree of a vertex*
The number of colored parents incident on the vertex v.

To achieve the proven objectives in an efficient manner, the **Vertex Coloring** approach was achieved using a bit vector implementation.

This is shown below in Figure: 1.

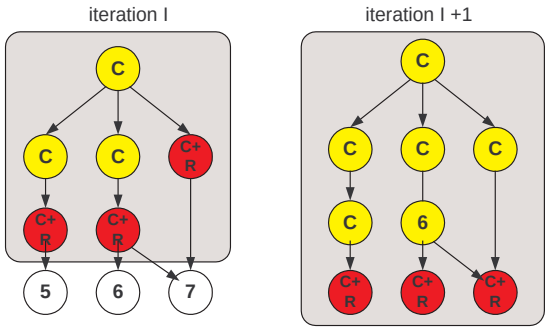

Figure 1: Vertex coloring in two consecutive iterations of BFT

3.2 SuperNode concept

As per the previous discussion,at each level,the NSP issues would be resolved Iteratively.Inorder to identify and group non-interdependent task nodes at each level,the concept of ***supernode*** is proposed.Each supernode is an encapsulation of task nodes with the following properties.

- **There are no interdependencies between the task nodes in a supernode.**
  In Figure:2,there are no interdependencies between nodes 2, 3, 4. Therefore they are grouped into one supernode. But in Figure: 3 since there is a dependency between nodes 3 and 4 , node 4 would not be considered for the super node with node 3 , even though it lies in the same depth as 3 from the root.
- **There is at least one common parent between every task node in a supernode** .
  Therefore nodes with disjoint parent sets would be in different task nodes. In Figure: 3, task nodes 6, 7 have common parents 1, 2, 3.therefore they are grouped to a single super node. But the nodes 8, 9, 10 have common parents 4, 5. Therefore they are grouped into another supernode.This facilitates in identifying disjoint tasks that can be run in parallel, without reducing the parallelism.

In order to construct the disjoint supernodes in an efficient manner, the **Disjoint-Set Forest** implementation[4] for disjoint sets , we used the "Union By Rank" and Path Compression" heuristics. The basic operations MAKESET(V) ,UNION(v_1,v_2) and FIND-SET(V) have been implemented inline with the supernode concept.

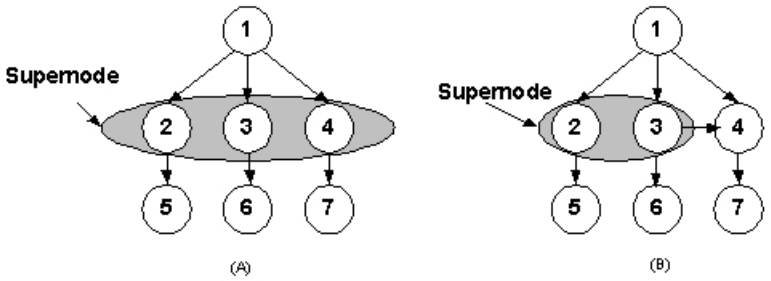

Figure 2: task nodes in a Supernode

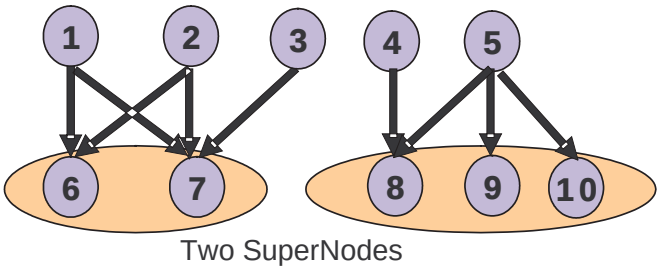

Figure 3: disjoint supernodes

3.3 Promoting a Sync node

After creating the supernodes, the NSP problem(s) in each supernode would be generally addressed at each supernode level (except in the case of a disjoint sync node, which is addressed later). While solving the NSP issues at each supernode, we select one of the sync nodes , from the set of sync nodes in the supernode, and promote it as the sync node for the rest of the siblings in the supernode.The procedure is as follows:

- Let all the un-synced parents of all the task nodes in the considered supernode sync to the selected sync node.
- All the other siblings in the supernode would be spawned from this sync node. This would result in adding of an additional synchronization level to the SP graph. The key advantage is that no additional synchronization nodes are added, thus from task node perspective, T_1 is not increased.
  This is shown diagrammatically in Figure: 4.

The following proves that this strategy would not violate any existing precedence relationships.

Lemma 3 *Among the candidate sync nodes* $\{V\} \in$ *Supernode s, (if any), let* $v' \in \{V\}$ *be a randomly chosen sync node. Then,*

- *All parents* $\{P\}$ *of* $\{V\}$ *can be synced to v'*
- $\forall\, v \in \{V\} - v'$ *can be safely spawned from the randomly chosen sync node.*

without violating the existing precedences.

Proof There does not exist a dependency between the nodes in a supernode(by definition).

$$\forall\, v_i, v_j \in Supernode(s), \neg\exists\ edge(v_i, v_j), where\ i \neq j$$

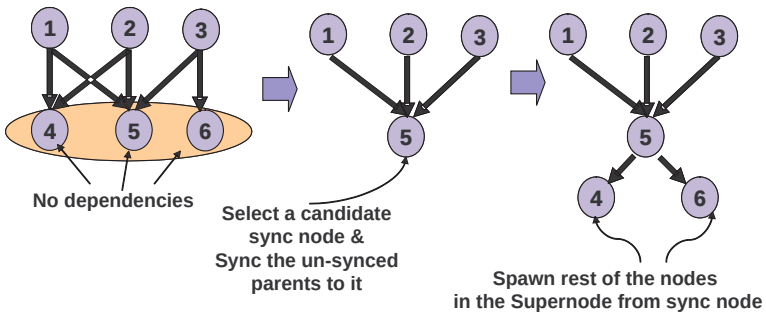

Figure 4: Promoting a sync node

Therefore one of them can be randomly chosen as a sync node, and the others can be spawned from this chosen sync node. However, the chosen sync node, acts as the new sync node for all the parents of the sibling nodes in the supernode. Since, the parents of all the sibling sync nodes, have synced to the chosen sync node, the spawning of the the sibling nodes does not violate the precedence relation. The spawning of the siblings from the chosen sync node leverages the parallelism in the execution of the nodes in the supernode. If v' is the randomly chosen sync node, all the other siblings $\{V\} - \{v'\}$ can be spawned from v'. □

Further, a special kind of NSP problem is the *transitive relationship*, as explained in Appendix A. Our solution to this, is to select the sync node in topological precedence as the only sync node of the spawn node in question, without violating existing precedences. The following lemma proves its validity.

Lemma 4 *If a vertex $v' \in G$, has distinct paths P_i and P_j ($P_i \neq P_j$) to more than one sync node v_i and v_j in G and there is a path from v_i to v_j , such that $Level(v_i) \leq Level(v_j)$.This problem is resolved by sync'ing v' to the sync node v_i that precedes all the other sync nodes v_j.*

Proof From the above definition, Level(v') ¡ Level(v_i) < Level(v_j) .Since ∃ distinct paths from v' to v_i, v' to v_j and v'_i to v_j, there is a indirect precedence between v' and v_j.Therefore this resembles an serial execution sequence $v' \succ v_i \succ v_j$. Hence without violating the existing precedence it is sufficient to model the dependencies between v' , v_i and v_i , v_j. This can be achieved by letting v' sync to v_i and preserving the dependency $v_i \succ v_j$. □

Now, extending form the above proof we will look at a more general case where, even when the two sync nodes does not have any transitive relationships, even though they are in different levels.

Claim 1 *If a vertex v' in SP DAG G' has already been synced to a sync node v_i in G', v' should not have an edge to another sync node v_j in G', where $Level(v_i) \leq Level(v_j)$.*

According to lemma 2, all vertices at level L_i are resolved before vertices at level L_j where $L_i < L_j$. Therefore when the BFT is processing level L_i, the sync node at level L_j would not be considered for processing. This precedence would model a execution sequence of $v' \succ v_i \succ v_j$. Therefore as per lemma 3, a candidate sync node would be selected at level L_i which might be v_i. Also, since there is no dependency between v_i and v_j there can be another vertex v_k in the same level L_i where ∃ precedence relation between v_k and v_j and ∃ a path from v' to v_j through v_k. Then from lemma 3, all parents of vertices at level L_i including v' would sync to the sync node, which might be v_i or another. Lets look at this in two scenarios.
Case 1: When the sync node at level L_i is v_i.
By Lemma 3, all the parents including v' would sync to v_i and the rest of the siblings of v_i would spawn form v_i.Then the precedence relation between v' and v_j would be $v' \succ v_i \succ v_j$ through v_i. Therefore there is no need to represent the precedence $v' \succ v_j$ again.Thus the precedence holds.

Case 2: When the sync node at level L_i is not v_i.
In this instance,another node v_{other} would be selected. Thus as per lemma 3,all parents including v' would sync to v_{other} and all siblings including v_i , v_k would spawn from v_{other}. Then the precedence relationship would be $v' \succ v_{other} \succ v_i$ and $v' \succ v_{other} \succ v_k \succ v_j$.Since v_{other} is the only sync node for v' as per lemma 3 and the precedence relation $v' \succ v_{other} \succ v_k \succ v_j$ there is no requirement for additional precedence relation representation for the relation between v' , v_i and even v' , v_j.
Therefore it proves the fact that, if a vertex is already synced to a vertex at level L_i it is not required to sync it again to a node at level L_j where $L_i < L_j$.

3.4 Synchronizing to a Disjoint sync node

Definition 20 *Disjoint sync node(DSN)*
A node is a disjoint sync node if any of its parents are disjoint vertices.

This is an instance where there are sync nodes at the current level, descending from parents that have different spawn nodes, which had distinct execution paths before the current level. This is shown in the LHS of Figure 5. Before delving to our transformation strategy, let's try to prove that there were multiple parallel execution paths before the current level that ultimately lead to the disjoint sync node.

Lemma 5 *Let $\{S_i\}, \{S_j\}$ be the spawn sets of vertices v_i, v_j respectively. If $\{S_i\}, \{S_j\}$ are disjoint sets i.e. $\{S_i\} \bigcap \{S_j\} = \phi$, then v_i, v_j can have different execution paths until they sync up to sync node s such that, $Level(s) > Level(V_i)$ and $Level(s) > Level(v_j)$.*

Claim 2 *The vertices v_i, v_j have parallel execution paths.*

Proof Let G_i and G_j be the sub-graphs rooted at the disjoint vertices v_i and v_j. Since the spawn sets maintain the spawn vertices from which the vertex v_i was spawned, if they are disjoint, then there does not exist a dependency between the vertices v_i and v_j. Therefore subgraphs rooted at v_i and v_j can be executed in parallel until a sync node is encountered, or if there exists a dependency between the vertices in the sub-graph. The execution paths are disjoint as long as no dependency exists between the sub-graphs or a sync node is encountered. In either of the cases, the disjoint spawn-set vertices v_i and v_j can be synched to the common sync point, until which parallel execution can proceed. □

Then we propose the following synchronization mechanism to resolve the NSP problem when there exists atleast one Disjoint sync node.

Proposition 1 *If there exists a disjoint sync node d, in one of the supernodes $s_i \in \{S\}$ for particular level* l *and let $\{V\}$ be the set of all the vertices at level* l*,then we propose that we would select d as the sync node for level l, and all the vertices $P \in G'$that are not synced yet, to be synced to disjoint sync node d.*
Further we would propose that all the vertices $v \in (V - d)$ could be spawned from node d. Both of these operations would not violate the existing precedences in the NSP DAG G.

Claim 3 *We claim that this proposition would not violate any existing precedence relationships in the original NSP DAG.*

Proof First this is a simplistic work around for preserving the precedence when multiple disjoint paths are synced.When disjoint paths are synced, there can be multiple sync structures aggregated together ,thus making the graph irreducible to a SP structure . Therefore a less complex way of preserving the precedence is, letting all the vertices P ∈ G' that are not synced yet (to preserve one sync node constraint for each task node - lemma 4), to be synced to *d*. Further from lemma 2, there are no inter-dependencies between the vertices in {S}. Therefore spawning vertices v ∈ $\{S - d\}$ from *d* would not violate any dependencies. Also we know that levels of all the vertices ∈ {S} are less than all the vertices that are already resolved(lemma

1). Thus we claim that this would not violate the precedence, thus the claim is valid. □

Further, since the disjoint sync node is the only sync node for the considered level,if there are transitivity relatioships between vertices that lie above and below the disjoint level needs to be resolved. The following lemma addresses this issue.

Lemma 6 *For all vertices v, and disjoint sync node d, let Level(v) ¿ Level (*d*). For all the ancestors of v, who have an edge to v, which topologically precede d, the edges from the ancestor to v can be expunged in G'.*

Proof We know that, for a disjoint node, all the nodes that topologically precede the disjoint node are definitely synced to the disjoint-sync node [proposition 1]. Thus in every topological ordering of the nodes, the ancestor which is topologically ordered before the disjoint-sync node would always be synced to the disjoint-sync-node. Hence, even by omitting the parent-child node link, we maintain the graph properties of precedence by the Contra positive rule of transitivity.

Contra positive of the rule of transitivity:

$$\text{Level(Ancestor)} < \text{Level(Disjoint-sync-node)} = \begin{cases} If Level(Ancestor) < Level(disjoint - sync - node) \\ \quad and \\ Level(Disjoint - sync - node) < Level(v) \end{cases}$$

Let's look at it in the 2 possible scenarios:

Case 1:
A vertex v such that $Level(v) > level(D)$ and $Level(D) > Level(Parent(v))$. As proven earlier, all nodes v' where Level(v') ¡ Level(D) are directly or indirectly synced to D. Then in the original NSP the edge(Parent(v), v) would be in transitive relationship with $\{(Parent(v), D) \cap \{(D, v)\}$. Therefore this results in an redundant dependency edge(Parent(v), v). Therefore the edge dependency {(Parent(v), v)} can be expunged.

Case 2:
The vertex v with parents v_i *and* v_j, such that $Level(v_i) > Level(D)$ and $Level(D) > Level(v_j)$. There is a dependent edge relationship like: $(D \rightarrow v_k \rightarrow v_{k+1} \rightarrow v_{k+2}...v_{k+n} \rightarrow v_i)$.

As proven earlier, all nodes n such that $Level(n) < Level(D)$ are directly or indirectly synced to D. The when the two dependency edges for v, $v_i \rightarrow v$ and $v_j \rightarrow v$ are considered, there is a transitive relationship between $v_j \rightarrow D \rightarrow v_k \rightarrow v_{k+1} \rightarrow v_{k+2}...v_{k+n} \rightarrow v_i$. and $v_j \rightarrow v$. Therefore the two transitive relationships can be reduced to a single $edgev_i \rightarrow v$ due to the fact that the other edge is already synched to a previous level. Therefore since the precedence relationship between $v \rightarrow v_i(Parents(v))$ and D are preserved as it is in the NSP DAG, the disjoint set creation and link removal preserves the correctness of the graph. □

Now lets examine the following example. As per the figure below, The sync node 9 has parents 5,6 that have different spawn nodes. Thus node 9 would be considered as a ***disjoint sync node***. We can see that even though nodes 8, 10 have different parents, that result in different supernodes, the parents of 8, 10 are also synced to node 9. Further, the nodes 8, 10 are spawned from node 9. Even though this results in an reduction in parallelism, it ensures a proper SP synchronization structure for the level.

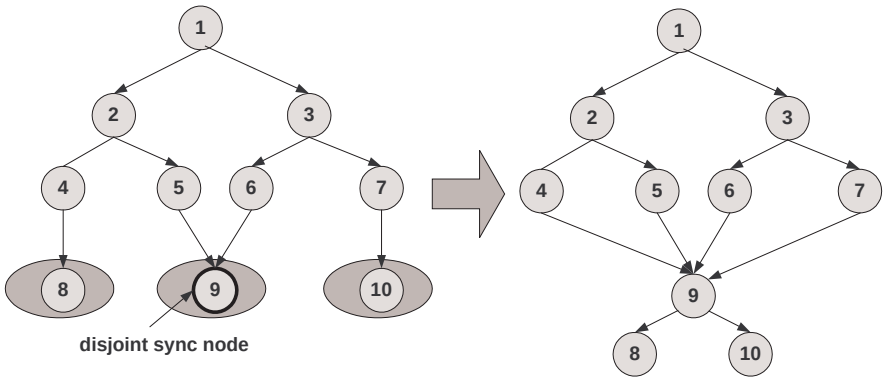

Figure 5: sync to a disjoint sync node

3.5 Flow of the Algorithm at a high level

After delving on the basic concepts behind the algorithm, following flow chart shows the overall algorithm at a high level.

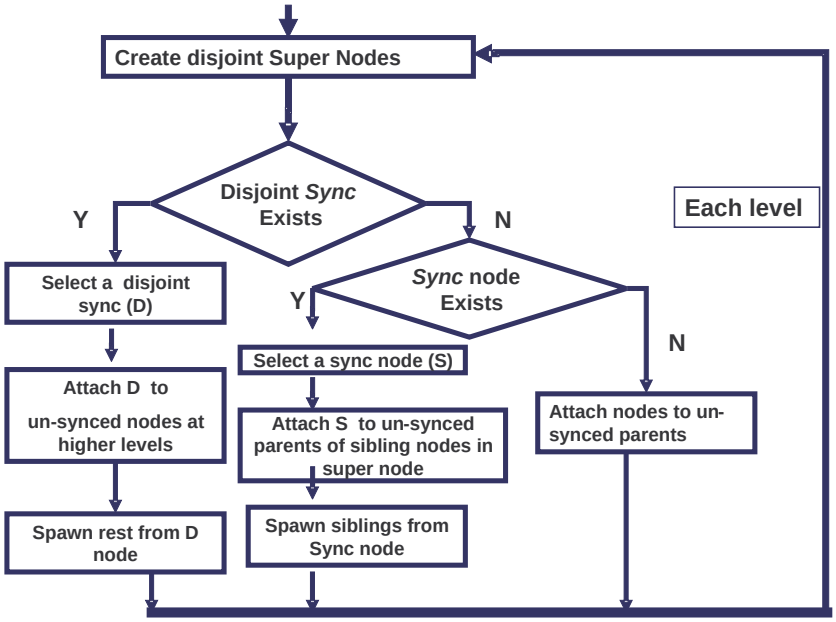

Figure 6: Flow of the Algorithm

4 Algorithm

This section looks at the intricate details of the algorithm and its implementation mechanism.
Description:
This is the main entry point to the algorithm.This performs the breadth first traversal.

```
BFT(NSP − DAG G)
1  for ∀ level L_i of the NSP DAG G
2       do
3          let V ← set of all vertices at level L_i
4          CreateDisjointSets({V})
5          let {S} ← supernodes created from CreateDisjointSets
6          Convert-SP({S})
7  End
```

Description:
Creates the set of disjoint supernodes that correspond to the disjoint sets of vertices for the level L_i during the BFT. The algorithm is based on the disjoint-set forest algorithm with the following heuristics.
1. **Union by rank** - During UnionSuperNodes (S_1,S_2) of two super nodes, the root with fewer nodes is pointed to the root of the tree with more nods. Rank of a super node is denoted by rank[S].
2. **Path compression** - During FindSuperNode(v), make each node on the find path in the disjoint forest point directly to the root without changing any ranks. Let P[v] denote the parent of v in the disjoint-set forest.

```
CreateDisjointSets (Vertices {V})
 1
 2  {V_all} ← {V} ∪ Parents({V})
 3  for ∀ v ∈ Parents({V})
 4       do MakeSuperNode (v,false)
 5  for ∀ v ∈ {V}
 6       do MakeSuperNode (v,true)
 7  for ∀ edge p → v where p ∈ Parents({V}) and v ∈ {V}
 8       do
 9          S_v ← supernode of v
10          S_v ← supernode of p
11          {S} ← UnionSuperNodes (S_v,S_p)
12  for ∀p ∈ Parents({V})
13       do
14          remove p
15  for ∀ v ∈ {S}
16       do addParentSpawnVector(v)
17
18  End
```

Description:
Creates a supernode s, with the representative Vertex v with a rank 0.

MakeSuperNode ($Vertex\ v, bool\ isChild$)

```
 1  s ← ⊘
 2  D bit of s ← 0
 3  if ( v isChild)
 4      then
 5           if ∀ parent of v's C bit ==1
 6             then
 7                  S ← S ∪ v
 8                  D ← getDisjointParentStatus(v)
 9                  if (D)
10                    then
11                         S bit of S ← 1
12
13
14      else
15           if (C-bit of v =1)
16             then
17                  S ← S ∪ v
18  rank[ s] ← 0.
19  Return s.
```

Description:
Unites two supernodes p and v, where the supernodes v,p are disjoint prior to the union generation.

UnionSuperNodes ($supernode\ p, supernode\ v$)

```
 1  S_p ← FindSuperNode(p)
 2  S_p ← FindSuperNode(v)
 3  Let D_p be the D bit of supernode p
 4  Let D_v be the D bit of supernode v
 5  Let D be the D bit of joined supernode.
 6  if (rank[ S_p ] > rank[ S_v ])
 7      else
 8           P[ S_p ] ← S_v
 9  if (rank[ S_p ] == rank[ S_v ])
10      then rank[ S_v ] ← rank[ S_v ] + 1
11  D ← Bitwise-OR(D_p,D_v)
12  End
```

Description:
Finds the supernode where the vertex v is represented currently

FindSuperNode($Vertex\ v$)

```
1  if (v ≠ P[v])
2     then P[v] ← Findset(P[v])
3  return P[v].
```

Description: Resolves the vertices V ∈ {S} and adds them to the resultant SP graph G'.

```
ConvertSP(SuperNodes {S})
 1  Let v_i be the disjoint sync node returned by Findsync
 2  for ∀ vertex v ∈ {S}
 3        do
 4           If ( Parents(v) == ∅)
 5              then G' ← G' ∪ v.
 6                   Bit Vector of v ← < 1, 1 >
 7              else
 8                        do
 9                           If(∃ disjoint supernode d ∈ {S})
10                              then
11                                   Let s' ← first supernode ∈ {S}.
12                                   v' ← findsync(s',isDisjointNode)
13                        else
14                                 do
15                                   for ∀ s' ∈ S
16                                   if ( s_s = findsync(s',isDisjointNode) ≠ ∅)
17                                     then
18                                          AttachSyncNode(s_s,s)
19                                     else
20                                          for ( ∀ vertex v ∈ s')
21                                          AttachNodeToSP(v,Parents(v))
22  End
```

Description: Finds and returns a random sync node from a supernode ***s***.

```
FindSync(Supernode s, bool IsDisjointNode)
 1  S_d ← GetDisjointSetSyncNode(S)
 2  if S_d ≠ φ
 3     then SetGlobalLevel
 4          IsDisjointNode ← TRUE
 5          return S_d
 6  elseif ∃ v ∈ S
 7  if GetAugmentedInDegree(v) > 1 and Level(Parent(v)) > GlobalLevel
 8     then Select the first candidate S_y.
 9          Return S_y.
10     else
11          Return null                    ▷ No sync nodes were found
12  End
```

Description: Attaches the selected sync node to the parent vertices of all the vertices in the particular supernode.

AttachSyncNode($Vertex\ v, Supernode\ S,$)

```
1  v' ← { Parents(v), such that v ∈ S and R − bit(Parents(v)) = 1}
2  if Logical-OR(R-bit(Parents(v))) = 0
3     then
4          v'' ← v' ⋃ Parents(v) where R − bit(Parents(v)) = 0
5          AttachNodetoSP(v, v'')
6  elseif Logical-OR(R-bit(Parents(v))) = 1
7     then
8          v'' ← v' ⋃ Parents(v) where R − bit(Parents(v)) = 1
9          AttachNodetoSP(v, v'')
10 if {S − v} ≠ φ
11    then V_p ← {S − v} ∈ S.
12         ∀v_i ∈ V_p   AttachSyncNode(v_i, v)
13 End
```

Description: Attaches a disjoint sync node to all the vertices with out-degree 0 of the SP DAG G', including the parents of all the vertices in the current level.

AttachDisjointSyncNode($Vertex\ v, Supernodes\ \{S\},$)

```
1  Let {V'} be all the predecessors P_i of all the vertices v in all the supernodes {S}
   such that R − bit(p_i) = 1.
2  if Logical-OR(R-bit(Parents(v))) = 0.
3     then v'' ← V' ⋃ Parents(v) where R − bit(Parents(v)) = 0
4  elseif Logical-OR(R-bit(Parents(v))) = 1
5     then
6          v'' ← v' ⋃ Parents(v) where R − bit(Parents(v)) = 1
7          AttachNodetoSP(v, v''). .
8  Let {V''} be all the vertices in all supernodes {S} excluding the disjoint node v.
9  if {V''} ≠ φ
10    then AttachNodeToSP(v, V'') ▷ attach the rest of the vertices ∈ {V''} to v
11 End
```

Description: Sets the global disjoint level of the DAG G', to the level of Vertex v.

SetGlobalLevel($Vertex\ v$)

```
1  GlobalLevel ← Level(v)
2  End
```

Description:
Finds the first candidate disjoint sync node (if any) d out the vertices in supernode s.

GetDisjointSetSyncNode($Supernode\ s$)

```
1  for ∀ candidate sync nodes v ∈ S
2       do
3          if GetDisJointParentStatus(v) is TRUE and Level of at least one Parent(v) > GlobalLevel
4          Then return v
5  End
```

Description:
Returns the status on whether the parents of a particular sync node have different spawn sets.

```
getDisJointParentStatus(Vertex v)
1  Let spawn vector of v ← ⊘
2  for ∀ parents p of v
3        do
4           if (spawnVector of p ≠ non-empty spawnVector of v)
5             then
6                   return TRUE
7             else
8                   spawnVector of v ← spawnVector of p ∪ spawnVector of v
9           Return FALSE.
```

Description:
Returns the number of parents of a vertex v ,that have already been resolved.

```
GetAugmentedInDegree(Vertex v)
1  For Parents(v)
2        do
3           if C = 1
4             then degree ← degree + 1
5  return degree
6  End
```

Description:
Adds a vertex v to the SP DAG G'.

```
AttachNodetoSP (Vertex v, Parents {P})
 1  Let {P_old} be the original parents of v in G.
 2  if GlobalLevel = 0
 3     then Set the edge p_i → v for parents p_i ∈ {P}.
 4          ∀ p_i ∈ {P} Set R = 0.
 5  elseif |P| = 1 and Level(P) = GlobalLevel.
 6     then Set the edge P_i → v.
 7          Reset R ← 0 for p_i.
 8     else
 9          for p_i ∈ {P}
10          if Level(P) > GlobalLevel
11            then
12                 Set the edge p_i → v.
13                 R ← 0 for p_i.
14  Set R ← 1 for v.
15  Set C ← 1 for v.
16  End
```

Description:
creates the spawn vector for vertex v .

```
AddParentSpawnVector(Vertex v, Parents {P})
1  for p_i ∈ {P}
2  if Outdegree(p_i) > 1
3     then spawnVector(v) ← p_i ∪ spawnVector(v).
4     else
5          spawnVector(v) ← spawnVector(p_i) ∪ spawnVector(v).
6  End
```

5 Proof of Correctness of the Algorithm

The proof of correctness addresses the following two issues.

- Preservation of precedence during the conversion from NSP DAG G to SP DAG G'
- SP properties of the SP DAG G'.

In order to prove that the generated graph G' is correct from the aspects of preserving precedence and SP property ,the following inductive approach is taken. Two functions are defined in the inductive approach. Let G be the original NSP DAG and G' be the SP DAG created at level L_i where 0<L_i<Depth(G). Let a precedence relationship between two vertices v_1,v_2 be denoted as **Pred(v_1, v_2)** where $v_2 \in$ Predecessor(v_1). Let's define functions to verify the precedence preservation and SP property.

- Let ***precedence(G,G',L_i)*** be the property;
  $\forall$ **Pred(v_1, v_2)** where v_1,$v_2 \in$ G and $v_1 \in$ supernode s $\in$ {S} at L_i and Level(v_1) = $L_i \equiv \forall$ **Pred(v_1, v_2)** , v_1,$v_2 \in$ G' and level(V_1) = L_i .
- Let **SPness(L_i)** denote that from level 0 to L_i (0< L_i < **Depth(G')**) in G' , the SP property has been guaranteed.

Now lets look at the induction based proof.

- **Base Case**
  Lets consider the level $L_i = 0$. Then as per lemma 1, 2 vertices {V} at a level L would be resolved before traversing to the next level L_2 (L< L_2). Therefore when L_i=0, the root node v at level 0 would be resolved to G' . Also since node count =1 and **Pred(*root*$_1$, *root*)** = root , ***precedence(G,G',0)*** holds. Further from the first definition of a SP DAG, G' is a SP DAG. Thus **SPness(0)** holds.
- **Induction Hypothesis**
  Let assume that at level $L_i(0 < L_i < Depth(G))$ the two functions hold. Let them be $SPness(L_i)$ and **precedence(G,G',L_i)**

  Now let's consider G where level =L_{i+1} $(0 < L_i < Depth(G))$.
  In order to ascertain the correctness at LevelL_{i+1} , all different scenarios of graph transformations that could be applied in a particular level is considered, case by case.

  - **Case 1**
    There are no sync nodes in the supernode

    Proof
    Lets consider **precedence(G,G',L_{i+1})** first. We know **precedence(G,G',L_i)** holds. Now as per Lemma 2 ,vertex at a level $Li + 1$ will only be added when all the resolved vertices v at L_i are resolved and added to G'. Therefore $\forall$ **Pred(v_1, v_2)** for nodes V_2 in level L_i and nodes in L_{i+1} in G' hold the same as for G after resolving. Therefore **precedence(G,G',L_{i+1})** holds.
    Also since there are no sync nodes, all the vertices at level L_i would be parallel to each other. From the third definition of SP DAG, , task nodes that are in parallel constitute a SP structure. Thus **SPness(L_i)** also holds.
  - **Case 2**
    ***There exists at least one sync node v' out of all the nodes V at level L_{i+1} in a particular super node* s.**
    Proof
    At each supernode s where there are sync node(s) v', as per lemma 3, every parent of all the vertices of s are synced to a selected node v'. Also the vertices $\in$ {s - v'} are spawned from vertex v'. Since there are no dependencies among v' and vertices in (s - v') (lemma 2) there are no precedence relationships among v' and vertices in (s - v') . Therefore as per lemmas 3, moving the vertices (S - v') below v' as spawn edges, does not break the existing precedence even

though it creates a new synchronization structure. Further every parent of vertices in s, is synced to one and only vertex v'. Thus selecting a sync node and promoting it as the only sync node for every vertex in S does not violate the existing precedence for Level L_{i+1}. Also we know from the induction hypothesis that **precedence(G,G',L_i)** holds. Therefore **precedence(G,G',L_{i+1})** holds.

Now let's look at the proof of SP property , which would be analyzed in 3 aspects.

- **Case 2.1** There are more than one sync node v at level L_{i+1} in supernode s, and there are vertices $\{P\}$ where $\{P\} \subset$ *Predecessor(v)* who have already been resolved but not synched to any node s' where Level(s') $< L_{i+1}$.

  Proof

  AS per lemma 3, a single node v_s, from all vertices $\{V\} \in s$ is selected where all un-synced parents $\{P_s\} \subset \{V\}$ would sync to v_s . Then from lemma 4 there won't be any $p_s \in P_s$ that syncs to more than one Sync node. Therefore this transformation satisfies the SP property for level L_{i+1}. But From the inductive hypothesis, sp property is held up to level L_i. Then using inductive hypothesis we can claim that up to level L_{i+1},**SPness(L_{i+1})** holds.
- **Case 2.2**

  There are more than one sync node v at the considered supernode s and there are spawn nodes $\{P\}$ where $\{P\} \subset$ Predecessor(v) who have already been resolved. Then there exists some nodes $\{P_s\} \subset \{P\}$ that sync to a node $v \in s$ as well nodes $\{V'\}$ where Level($\{V'\}$) $<$ Level (v) and level($\{P_s\}$) $<$ Level(v') and there are paths from $\{V'\}$ to v.

  Proof

  As per the lemma 4 and proof of claim 1,a transitivity issue like this, for a spawn node $p_s \in P_s$ could be resolved by selecting one sync node s , where its level is the minimum out of all the sync nodes of p_s .Then the node p_s would have already been synched to s. Then the same spawn node would not sync to v. Therefore when node v is selected as the candidate node, the predecessor p_s would not need to sync to v, thus preserving SP property for level L_{i+1}. But From the inductive step, SP property is held up to level L_i. Then using inductive hypothesis we can claim that up to level L_{i+1},**SPness(L_{i+1})** holds.
- **Case 2.3**

  There are more than one sync node v at a considered super node s at L_{i+1}, and there are spawn nodes $\{P\} \subset$ Predecessor($\{V\}$) who have already been resolved. Then some nodes $\{P_s\} \subset \{P\}$, sync to a node $v \in s$ as well as node v' where Level (v') $>$ Level (v).

  Proof

  This is the inverse of case 2.2.Therefore vertices $\{P_s\}$ would sync to the candidate node v rather than v', where Level (v) $= L_{i+1} <$ *Level*(v') - (lemma 4 and proof of claim 1). Therefore vertices $\{P_s\}$ would only have one sync node, which removes any transitivity issues.

  Further when the resolving level equals level (v') $> L_{i+1}$, then $\{P_s\}$ is already synced to v. Therefore from case 2, it would not sync again to v'. Thus at level L_{i+1} the SP property is preserved. But From the inductive hypothesis, SP property is held up to level L_i. Then we can claim that up to level L_{i+1},**SPness(L_{i+1})** holds.

- **Case 3**

  There are more than one sync node V at level L_{i+1} and at least one is a *Disjoint sync node.*

  Proof

  Let v' be a vertex in super node $s \in \{S\}$ at level L_{i+1}. When a sync node v' at level L_{i+1} is disjoint , then as per the proof of claim 3 and lemma 5,

  - All the parents of all the vertices in all the supernodes at L_{i+1} and
  - All predecessors of nodes at L_{i+1} which haven't being synced in G'

  would be synced to the disjoint node v. Thus as per previous cases, since there is only one sync node at level L_{i+1}, SP property is preserved and **SPness(L_{i+1})** is held.

Further ,from the proof of Claim 3, at level L_{i+1} precedence is preserved.Therefore as above the precedence relationships up to level L_{i+1} is held.

6 Complexity Analysis of the Algorithm

The complexity analysis is done in a breadth wise manner.
For the worst case ,lets assume the following.

- The DAG G = (V,E) with |V| = n and |E| = m, is densely connected. i.e Each level is a fully connected bipartite sub graph.Then from graph theory[5] we could assume that $|V| \approxeq |E|^2$.
- Average number of nodes at each level = b (Branching Factor). Therefore number of edges at each level = b^2.

At each level of the NSP DAG , the algorithm performs the following main functions :

- **Supernode creation**.
  The supernode creation is based largely on the disjoint set algorithm utilizing ***disjoint set forest*** implementation.At each level, both parents and current vertices would be considered in creating the disjoint sets.From [1] the worst case time complexity for m disjoint-set operations on n elements is O($m.\alpha(n)$).In the worst case , at each level , $2b$ elements would be considered (nodes at the current level + their parents). Therefore the number of operations m would be
  - *2b* MakeSuperNode operations.
  - *(2b-1)* UnionSuperNodes operations.
  - *2(2b-1)* FindSuperNode operations.

  This sums up to **m = 8b-3** operations.
  Time complexity for disjoint set operations :

$$\mathbf{O}(\textit{(8b -3)}.\alpha\textit{(2b)}) \tag{1}$$

  But each node would be checking for disjointness and coloring status of its parents in MakeSuperNode method. Therefore this would involve O(b^2). Hence the time complexity for supernode creation for each level is :

$$\mathbf{O}(b^2). \tag{2}$$

  Extending this to all the levels and n vertices, the loose upper bound time complexity for supernode creation for the task graph would be:

$$\mathbf{O}(n^2). \tag{3}$$

- **NSP to SP conversion**.
  As per the algorithm, at worst case the following complexities can be observed for each level.

  - **Disjoint sync node exists**. Then as per the algorithm, both FindSync and AttachDisjointSyncNode have upperbounds of $\mathbf{O}(b^2)$.
  - **Sync node exists**. Then both Findsync and AttachSyncNode have upperbounds of $\mathbf{O}(b^2)$.

  Therefore loose upper bound in conversion for each level would be:

$$\mathbf{O}(b^2). \tag{4}$$

  Then extending to all the levels in the NSP DAG ,as before the overall loose upper bound for NSP to SP conversion would be:

$$\mathbf{O}(n^2). \tag{5}$$

Considering (1) and (2) the overall Time complexity for the overall algorithm in converting a NSP DAG with n vertices would be:

$$\mathbf{O}(n^2). \tag{6}$$

7 Qualitative assessment of the transformation

7.1 Critical path increment

7.1.1 Upper bounds

The algorithm processes the NSP DAG in a breadth first manner. When the traversal depth is at level L_i , all the nodes {V} where Level(V) $< L_i$ are already resolved to a SP DAG. As per the definition of supernode, at each level L_i , all disjoint sub graphs of G , that belongs to different supernodes, would be resolved in parallel. Therefore irrespective of the number of disjoint sub graphs at level L_i , a sync node would be selected for synchronizing the NSP problems at level L_i for each sub graph.Therefore if there are n such supernodes, there would be one sync node promoted for each supernode , which would be in parallel to each other.Therefore irrespective of n, only one level of additional sync nodes would be added for each level of a NSP problem.
Further when there are NSP problems across disjoint parents of the nodes at level L_i (i.e disjoint sync nodes), irrespective of other NSP problems in the individual supernodes, the algorithm selects a disjoint sync node to synchronize all the nodes at level L_i (proposition 1), thus incrementing the critical path length by 1 for the level L_i.
Therefore at worst case, for a graph with a depth of L, for every levelL_i (0$< L_i <$ L) there can be at least one NSP problem. Therefore at each level, a specific synchronization is required. This results in a critical path increment of 1 * L .

- Original depth of the NSP graph = L
- Additional path length added in worst case = L

Therefore we claim that the upper bound in the increment of Critical path length is **2.L** for any type of NSP Graph with computational depth of **L**.

7.1.2 Lower Bound

For completeness, lower bound invariable would result in when an SP DAG is processed by the algorithm. Since there are no NSP problems at any level L_i (0$<L_i<$ L) that needs transformation, the graph would not be transformed.Therefore the lower bound would be **L**.

7.2 Increase in Critical path vs reduction in parallelism

When a task node is selected as an sync node for a level L to solve an NSP problem our algorithm creates a new level of synchronization in the task graph. Even though it does not increase the overall work (T_1) it increases the critical path T_∞.
As an example if we assume that there are ***n*** nodes at level L_i , selecting one candidate node as the sync node v would result in other ***(n-1)*** siblings spawning out from ***v***.The resultant would be;

- Increase in critical path by 1 .
- reduction in the parallelism at level $L_i$from ***n*** to ***(n-1)***.

In the case where this selected vertex ***v*** ,has a significant larger task load,then the rest of the ***(n-1)*** tasks get delayed until ***v*** is finished.Therefore as a variant to our existing algorithm we would like to explore the following heuristical approach.

7.3 Variant of the Algorithm using heuristics

If the loads of every task node in a supernode can be measured, we suggest the following heuristics in selecting the sync node rather than selecting a first sync node as the synchronization node.
First let's define cost SOH_c -As the overhead incurred in adding a new task node to G'.

Assumption 1 *Let's assume that the synchronization cost of the parent nodes on a sync node would be similar ,when either an existing task node or a new synchronization node is used. Thus is not used in the comparative cost analysis below.*

The suggestions are as follows;

- Select the sync task node v_{lo} with the lowest work load , say Vc_{lo} where cost(v_{lo}) $\leq \forall$ vertex $v \in$ current supernode .This would result in a lesser increment to T_∞ than selecting an arbitrary node, from the point of actual execution time of the Critical path inline with the definition in [2].
- Include a new synchronization node v_s where all parents of all vertices in supernode would sync to, when $Vc_{lo} \gg SOH_c$ .The following remarks needs to be considered.
  - This would result in an lesser increment in the actual execution time since $T_\infty + Vc_{lo} \geq T_\infty + SOH_c$.
  - But would result in an increase in the Total work since the addition of SOH_c. The new Total work would be $T_1 + SOH_c$.

8 Experiments and analysis

In order to verify the correctness and the quality of conversion, from the aspects of increase in critical path, precedence preservation and SP properties ,several experiments were carried out. They were categorized under two aspects.

- Modelling of task graphs that represent actual applications
- Asses the conversion quality by using random NSP Graphs with different topologies and work loads.

8.1 Actual Task Graph modelling

In this empirical analysis , several graph structures were utilized.Further, to assess the quality we compared our transformations with a previous transformation algorithm suggested by Gemund et.al [6].

8.1.1 Fully connected Bipartite Graph

We achieved commendable results for this transformations.At each level of the graph with a NSP issue, we were able to achieve SP transformation by addition of one level. Therefore in a Bipartite graph with n levels , the maximum increase in the critical path was *2n*.

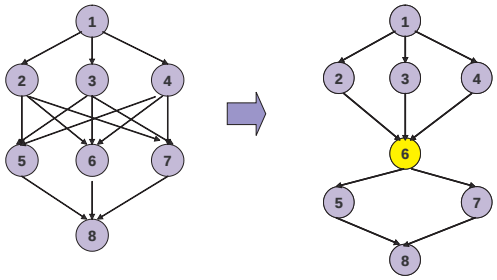

Figure 7: Bipartite graph transformation

The following graph shows the comparative results.

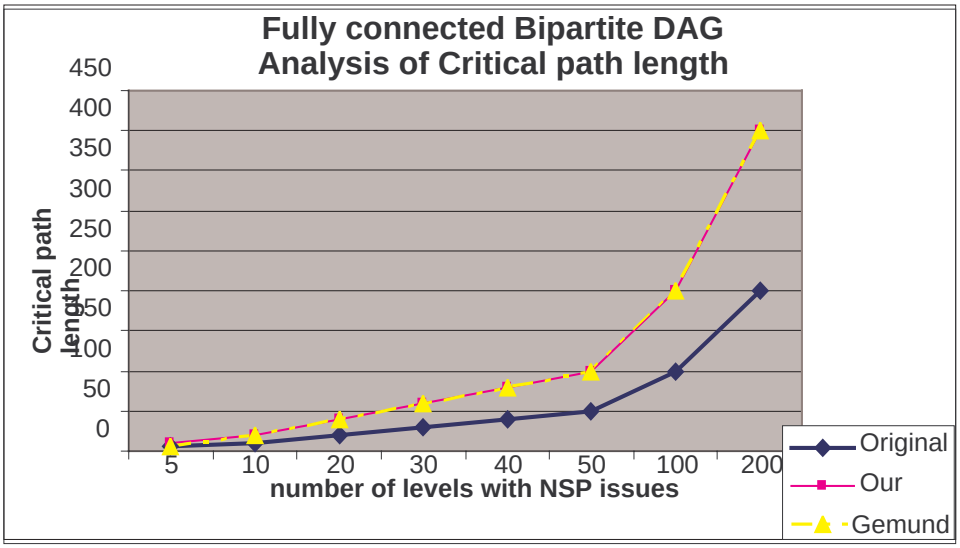

Figure 8: comparison of Bipartite graph conversion

8.1.2 Paired Synchronization Graphs

This can be modelled in parallelization of loops where a part of the loop body contains loop-carried dependencies.Our comparative results were very attractive.When we experiment out with NSP problem at only one level with multiple levels of SP subgraphs following it,our algorithm managed to narrow down the increment of T_∞ to 1. But critical path increment of [6] increased linearly proportionate to the number of SP levels below the NSP level as given in the figure 9.

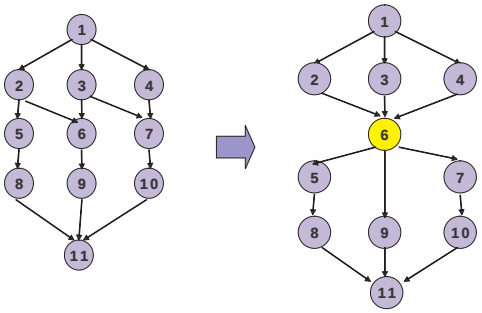

Figure 9: Paired Synchronization graph transformation

The following graph shows the comparative results.

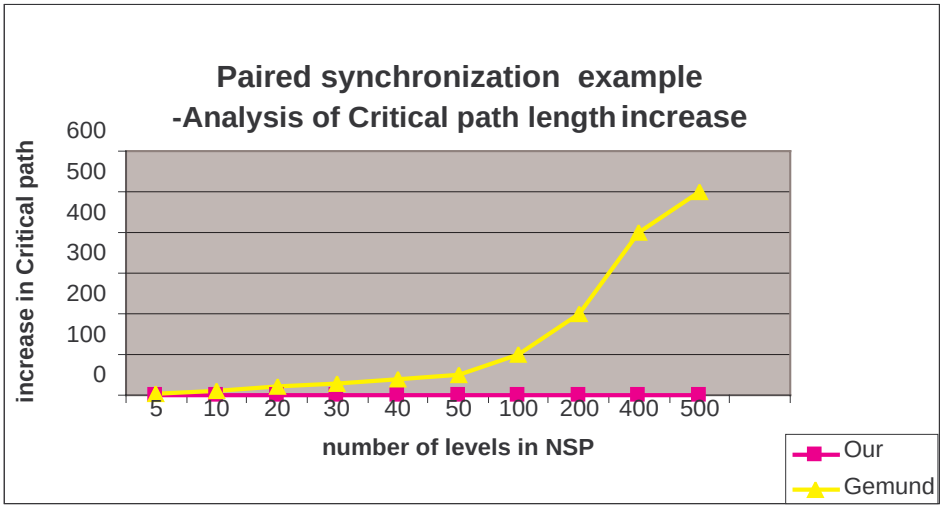

Figure 10: comparison of Paired Synchronization conversion

8.1.3 Macro Pipeline

The Macro Pipeline graphs can be seen in wave-front computation applications.Lets denote the width (and the breadth) of the task graph as $\boldsymbol{M}$, then our algorithm managed to restrict the Critical path increase to $\boldsymbol{2M\text{-}4}$, where as in the comparison [6] showed more increase in the critical path. Further there was no increase in the Total Work T_1. Following graph shows a comparison with [6][7]

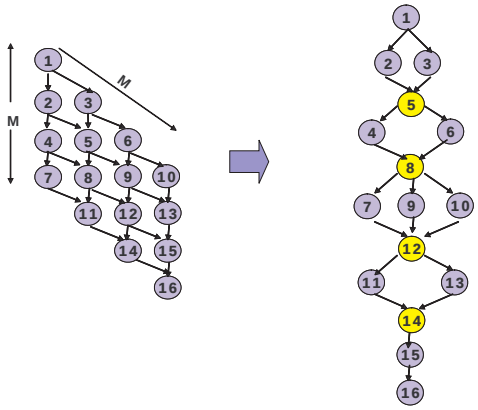

Figure 11: macro pipeline transformation

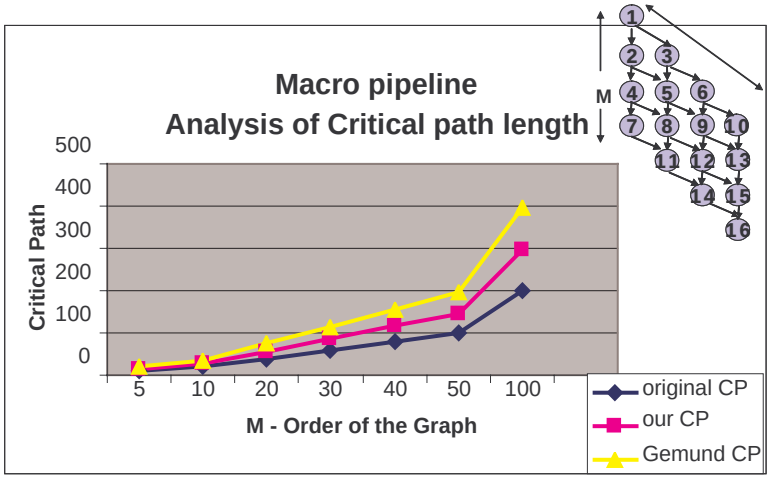

Figure 12: macro pipeline transformation

8.1.4 Blumofe's lower bound example [8]

This represents a NSP DAG with computationally deep and shallow tasks in the same portion of computation. The model in a multi-threaded computation would be as in Figure 13.The root thread spawns m threads of subcomputations. Then this could be represented in a NSP DAG as shown in Figure 14.

When we convert the NSP DAG to a SP DAG, we observed that the critical path increase was ***(m-1)*** where m denotes the subcomputations spawned from the root thread.Figure 15 shows the transformed SP DAG

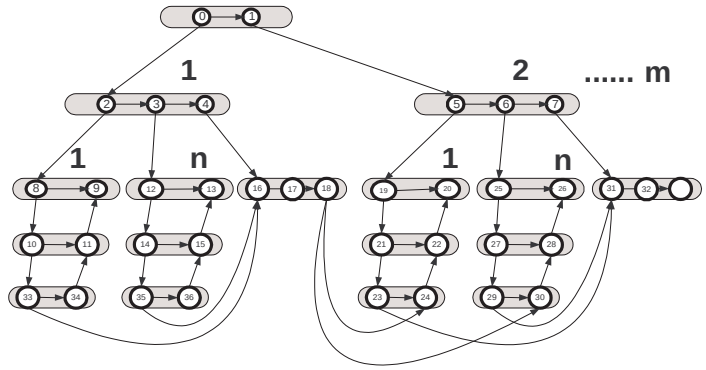

Figure 13: multi-threaded computation model

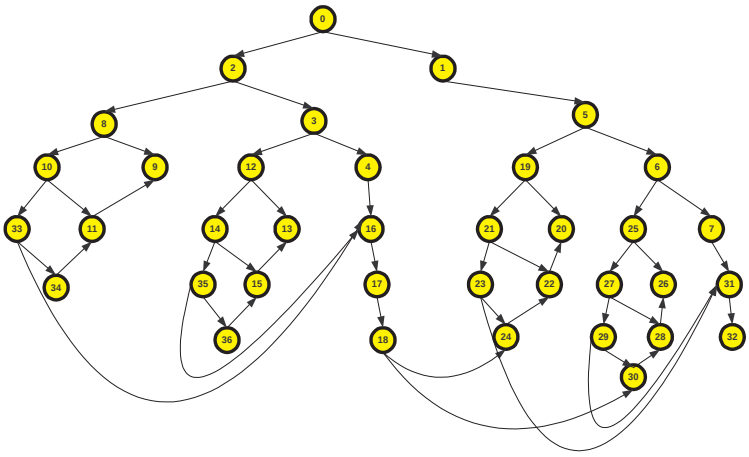

Figure 14: NSP DAG representation of the computation

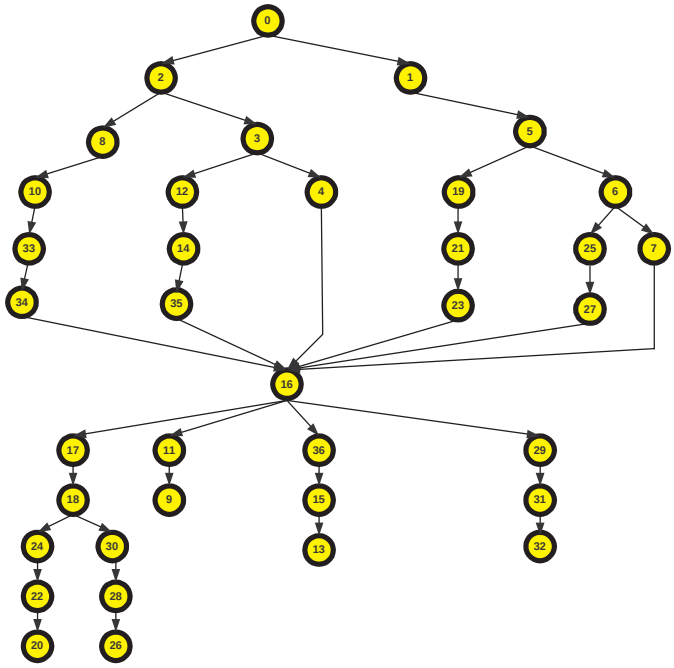

Figure 15: SP DAG representation of the computation

8.2 Random Graph Analysis

In order to verify the correctness and transformation quality of our algorithm,we utilized random Graphs generated by Waseda University , Japan [9].Two sets of graphs were utilized each having around 100 graph. One set comprised of 50 node task graphs while the other included task graphs of 100 nodes each. The Resultant graphs were verified for;

- Preservation of precedence
- SP property of the transformation.

Following shows the comparative results. The results generated from the random graphs were encouraging to the fact that , the increase in Critical path was always less than twice the critical path length of the NSP DAG. The summarized results for 100 node random task graphs were as follows ;

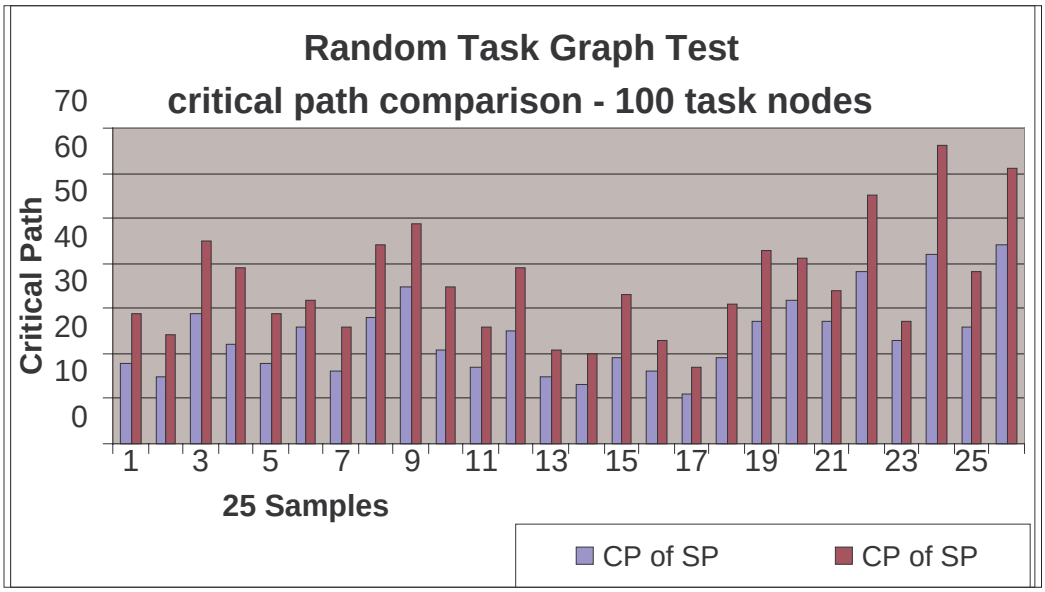

Figure 16: Results for 100 node random task graphs

Average Critical path length of NSP task graphs	23
Average Critical path length of SP task graphs	34
Maximum Ratio in CP increase	1.77
Minimum Ratio in CP increase	1.17
Increase in Total Work	0

Figure 17: Summarized results for 100 node random task graphs

8.3 Verification of the results

Keeping inline with the theoretical verification of the algorithm we took effort in verifying the quality of the transformation empirically. The following steps were taken.

- Verification of preservation of precedence in the transformation through a precedence checking program.
- Verification of SP properties through a SP property checker program.The Series-parallel digraph recognition algorithm proposed by Tarjan et. al [10] was implemented in order to verify the SP properties in the transformed task graphs.

9 Conclusion

We were able to successfully implement a new algorithm that gives a more qualitative NSP to SP conversion with a lesser time complexity compared to the previous findings. Further after implementing the algorithm,we managed to verify the correctness and the quality through converting actual NSP computation models and random task graphs. Following summaries our achievements.

- Upper Bound for increase in T_∞ is $2.T_\infty$.
- Upper Bound for increase in T_1 is 0.
- Time Complexity of the Conversion Algorithm is $\boldsymbol{O(n^2)}$

9.1 Future Work and Enhancements

Following describes the enhancements that could have been applied in the project and projects the future scope of any possible extensions.

- The current scope address the NSP DAG to SP DAG conversion from a theoretical standpoint.But the conceptual transformation can be verified by modelling the transformation in an actual parallel multi-threaded environment like *Cilk*.This would facilitate in analyzing the actual increase in T_∞ and T_1 according to the cost model of the underlying environment.
- The current algorithm results in unbounded in-degree when synchronizing levels with disjoint sync node.This results in a bottleneck for that particular level through the disjoint synchronization node. Another possible exploration area would be refine the algorithm to avoid such bottleneck.
- Even though we were able to perform the transformation with significant effects on the time bounds, we haven't performed a thorough analysis of the effects on space bounds after the transformation.This is essential since its would be fruitless to achieve significant time bounds with relatively inefficient space bounds.
- As there exists parallel algorithms and implementations to verify the "SP-ness" of a arbitrary DAG, a possible extension would be to delve into a parallel implementation of the algorithm.This would involve in analyzing possible vertical/horizontal partitioning of the task graphs to disjoint sets (similar to the Supernodes) and applying the current algorithm on those disjoint task graph sets.
- Further ,we would like to verify the correctness of the algorithm with more models of actual task graphs as well as random task graphs with more variety and task nodes. Further it would be relevant to verify the correctness using task graphs with different task loads rather than unit work loads.This would enable us to compare this with the variant we have discussed and analyst the pros and cons of each method.

10 Appendix

10.1 Appendix A - NSP problems

This section outlines the following NSP problems ,(Where any NSP problem could be decomposed into [6]) that are addressed in our Algorithm.

- A spawn node syncs to sync nodes that are in a transitive relation.The sync nodes are in different levels in the NSP DAG.This can be called a transitive relation problem.
- Spawn node(s) syncing to more than one sync node that are in the same level (e.g. simple bi-partite graph).

References

[1] V. P. Arturo Escibano, Arjan Gemund, "Performance trade-offs in series-parallel programming models," 1997.

[2] H. P. Charles E.leiserson, "A minicourse on multithreaded programming," 1998. Also available as MIT Laboratory for Computer Science.

[3] R. D. Blumofe, *Executing Multitreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report.

[4] R. C. C. E.Leiserson, T.Cormen, *Introduction to Algorithms.* (Second edition)MIT Press, 2001.

[5] J. A. McHugh, *Algorithmic Graph Theory.* Prentice Hall Publications, 1990.

[6] A. Escibano and A. Gemund, "An algorithm for transforming nsp to sp graphs," (Delft University of Technology Netherlands), 1996.

[7] A. G. Arturo Escibano, "An algorithm for transforming nsp to sp graphs," (Delft University of Technology Netherlands).

[8] R. D. Blumofe, "Managing storage for multithreaded computations," Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992. Also available as MIT Laboratory for Computer Science Technical Report.

[9] "Standard task graph sets -kasahara laboratory,waseda university." Available on the Internet from $http://www.kasahara.elec.waseda.ac.jp/schedule/intro_e.html$.

[10] J. V. Robert Tarjan, "The recognition of series parallel digraphs," in *Proceedings of the ACM.*