Computer Science Department
University of Valladolid
Valladolid - Spain

A New Algorithm for Mapping DAGs to Series-Parallel Form

Arturo González-Escribano[1], Arjan J.C. van Gemund[2], and Valentín Cardeñoso-Payo[1]

[1] Dept. de Informática, Universidad de Valladolid.
E.T.I.T. Campus Miguel Delibes, 47011 - Valladolid, Spain
Phone: +34 983 423270, eMail:arturo@infor.uva.es

[2] Faculty of Information Technology and Systems (ITS)
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786168, eMail:a.j.c.vangemund@its.tudelft.nl

Abstract. This report presents a new algorithm technique that transforms DAGs (Direct Acyclic Graphs) into SP (Series-Parallel) form. The complexity bounds are $O(m+n)$ in space and $O(m+n\log n)$ in time.

Technical Report No. IT-DI-2002-2

1 Introduction

In this report we present a new algorithmic technique that address the problem of SP-ization: transformation of a DAG (Direct Acyclic Graph) to two terminal series-parallel form (as defined in [1, 3, 6]) with no task duplication (work preserving), but only adding dependencies.

The algorithm purpose is to minimize the probabilities of critical path value increase due to new added dependencies, but using no knowledge of actual workload distribution. In this case, a fair assumption is to consider all task loads i.i.d. (independent identically distributed). Thus, an algorithm that does not increase the critical path of an UTC (Unit Time Cost per task) graph, has the better chances of success.

We will compare our algorithm with others: Simple layering (full barrier synchronization between node layers) and a previous less refined algorithm presented in [4].

This new algorithm present the following interesting features:

- A reduced time complexity: $O(m + n \log n)$ and space complexity $O(m + n)$, where m is the umber of edges and n the number of nodes of the input graph.
- Local resynchronization of minimum number of nodes, while global information is considered.
- It does not increase the critical path for UTC (Unit Time Cost) graphs, keeping the nodes layering structure of the original graph.
- The solution of the algorithm is the same for a given topology independently of the input order (node labeling).

2 Graph preliminaries

We present some basic notation used throughout the following sections.

2.1 Basic graph notation

Let $G = (V_G, E_G, \tau)$ be a DAG (direct acyclic graph) in which V_G is a finite set of nodes and E_G a finite set of edges (or ordered node pairs) and $\tau(v \in V_G)$ the workload distribution function that assigns to every node its load value. We work with *multidigraphs*, which allow several instances of the same edge (u, w) in E_G at the same time. Let $indeg(v \in V_G)$ be the number of edges (u, v) for which v is the *target*, and $outdeg(v \in V_G)$ be the number of edges (v, w) for which v is the *source*. $R(G)$ is the set of *roots* of G (nodes $v : indeg(v) = 0$), and $L(G)$ is the set of *leaves* of G (nodes $v : outdeg(v) = 0$).

An StDAG is a DAG with only one root and one leaf ($|R(G)| = 1, |L(G)| = 1$). Any DAG can be transformed to an StDAG by a simple algorithm that adds a new unique root connected to all original roots and a new unique leaf connected to all original leaves.

A *path* between two nodes ($p(u, w)$) is a collection of nodes $u, v_1, v_2, ..., v_n, w$ such that $(u, v_1), (v_1, v_2), ..., (v_{n-1}, v_n), (v_n, w) \in E_G$. The *length* of a path is

the number of edges determined by the path sequence. The *critical path value* ($cpv(G)$) is the maximum, over all possible paths in G, of the accumulated load along the path.

The *depth level* of a node ($d(v)$) is the maximum length of a path from a root to that node. The *maximum depth level* of a graph ($\hat{d}(G)$) is the maximum length of a path from a root to a leaf. We define *d-edges* as the subset of edges $(u,v) : d(v) - d(u) > 1$. We say that a node v is *previous* to w or w *depends on* v ($v \prec w$) if there is a path $p(v,w)$. We say that two nodes v, w are *connected* ($v \prec\!\succ w$) if v depends on w or w depends on v.

2.2 Series-parallel graphs

A *series reduction* is an operation that substitutes a node v : $indeg(v) = outdeg(v) = 1$ and its two incident edges $(u,v),(v,w) \in E_G$ for an edge (u,w). A *parallel reduction* is an operation that substitutes several instances of an edge (u,w) for a unique instance of the same edge (u,w).

The *minimal series-parallel reduction* of a graph, is another graph obtained after iteratively applying all possible series and parallel reductions. A *trivial graph* $G_t = (V_{Gt}, E_{Gt})$ is a graph with only two nodes and only one edge connecting them ($V_{Gt} = \{u_1, u_2\}, E_{Gt} = \{(u_1, u_2)\}$).

An StDAG is SP (Series-Parallel or Edge Series-Parallel) iff its minimal reduction is a trivial graph G_t [1, 6]. The equivalence of Edge Series-Parallel graphs and Node Series-Parallel graphs is presented in [6].

3 Algorithm description

The algorithm is based in a depth level search, solving non-SP problems while it traverses the graph. The already processed subgraph is SP. A tree representing its minimal reduction graph is used to help in the search for resynchronization handles, transitivity checks and operations that have lesser complexity bounds in a tree. Evaluation of edges that express dependencies across several layers is delayed until the targeting layer is processed.

3.1 Initialization phase:

i. Transform the input DAG into an StDAG along the lines previously discussed.

ii. Layering of the graph. Compute a partition of V_G, grouping nodes with the same depth level.

$$L_i \subset V_G; L_i = \{v \in V_G : d(v) = i\}$$

iii. Initialize an ancillary tree $T = (V_T, E_T)$ to L_0. This tree will represent the minimal series-parallel reduction of the step by step processed subgraphs.

3.2 Graph transformation:

For all layers (sorted) i from 0 to $\hat{d}(G) - 1$:

a. Split layer in classes of *relatives*: Let us consider the subgraph $S \subset G$ formed by $L_i \cup L_{i+1}$ and all edges from G incident to two nodes in this subset. We construct the partition of this nodes into connected subgraphs. We define *relatives* classes as the subsets of nodes that belong to the same connected component of S and the same layer, as in Fig. 1.

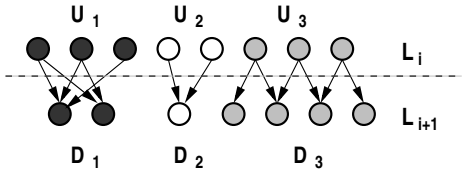

Fig. 1. Example of relatives classes induced between two layers

$\mathcal{P}_U = \{U_1, U_2, ..., U_n\}$ will be the *up* classes (of nodes in L_i) and $\mathcal{P}_D = \{D_1, D_2, ..., D_n\}$ will be the *down* classes (of nodes in L_{i+1}). Each class $U \in \mathcal{P}_U$ induces a class $D \in \mathcal{P}_D$ that belongs to the same connected component $(U \to D)$.

b. Tree exploration to detect *handles* for classes of relatives: We look in the tree for *handles*. For each U class, the *U-handle* $(h'(U))$ is the nearest common ancestor of all nodes in U:

$$H'(U) = \{v \in V_T : \forall w \in U, v \preceq_T w\}$$

$$h'(U) = h \in H'(U) : \forall h' \in H'(U) : d(h) \geq d(h')$$

We define $K_T(U)$ as the set of source nodes to the induced class D (it includes U and source nodes of d-edges targeting D): Sources of edges showing transitive dependency to D through the U-handle are to be discarded from $K_T(U)$:

$$K_T(U) = \{v \in V_T : (v, w) \in E_G, w \in D, v \not\preceq_T h'(U)\} \cup \{h'(U)\}$$

The *handle* node of class U, $h(U)$ is defined as:

$$H(U) = \{v \in V_T : \forall w \in K_T(D), v \preceq_T w\}$$

$$h(U) = h \in H(U) : \forall h' \in H(U) : d(h) \geq d(h')$$

We also define the forest of a class, as the set of complete sub-trees below $h(U)$ that include nodes in $K_T(U)$:

$$SubF(U) = \{u \in V_T : v \preceq_T u, (h(U), v) \in E_T, v \preceq_T w : w \in K_T(U)\}$$

In Fig. 2 we show a diagram of all concepts defined in this section.

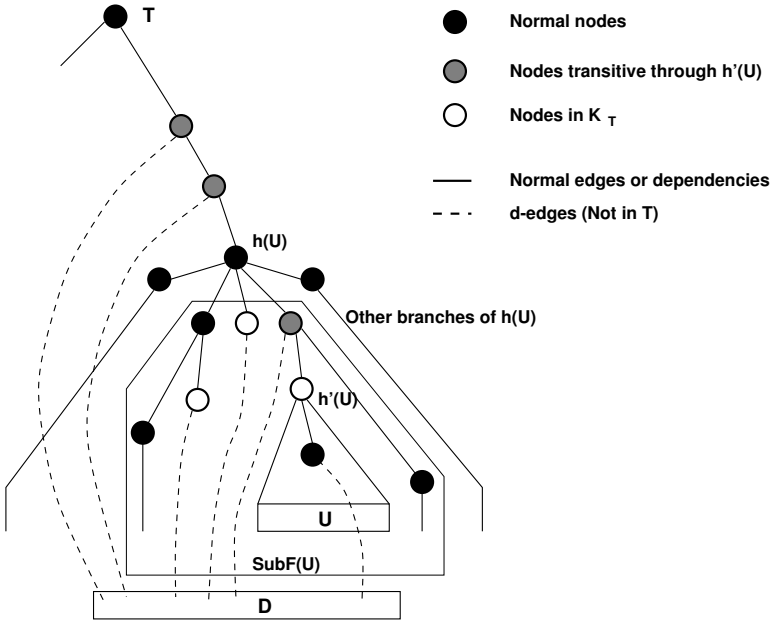

Fig. 2. Example of handles and forest for an U class

c. Merge classes with overlapping forests: Classes with overlapping forests are merged in an unique U and D class. They will be synchronized with the same barrier.

$$\forall U, U' \in \mathcal{P}_U : SubF(U') \cap SubF(U) \neq \emptyset$$

$$U = U \cup U'; \mathcal{P}_U = \mathcal{P}_U \setminus U'$$

$$U \to D, U' \to D'; D = D \cup D'; \mathcal{P}_D = \mathcal{P}_D \setminus D'$$

d. Capture *orphan nodes*: We define *orphan nodes* as the leaves of the tree T that are not in any U class (they are nodes in layers previous to i with only d-edges to layers further than $i+1$). These nodes are included in the U class of the forest they belong to.

$$\forall v \in SubF(U), v \in L(T), v \notin U; U = U \cup \{v\}$$

e. Class barrier synchronization: For each final $U \to D$ classes:

– Create a new synchronization node b_U in the graph and the tree.

$$V_G = V_G \cup \{b_U\}$$

$$V_T = V_T \cup \{b_U\}$$

– In G, eliminate all edges targeting a node in D. Add edges from every node in U to b_U and from b_U to every node in D (barrier synchronization).

$$E_G = E_G \setminus \{(v, w) : w \in D\}$$

$$E_G = E_G \cup \{(v, b_U) : v \in U\}$$

$$E_G = E_G \cup \{(b_U, w) : w \in D\}$$

– Substitute every d-edge (v, w) with source $v \in SubF(U)$ and targeting a node $w \in L_k : k > i + 1$ (a further layer) for an edge (b_U, w). This operation eliminate d-edges from the new synchronized SP subgraph, but avoiding the loss of dependencies in the original graph.

$$dE(U) = \{(v, w) \in G : v \in SubF(U), w \in L_k, k > i + 1\}$$

$$E_G = E_G \cup \{(b_U, w) : (v, w) \in dE(U)\}$$

$$E_G = E_G \setminus dE(U)$$

– Substitute the forest $SubF(U)$ in T for an edge $(h(U), b_U)$ representing the minimal series-parallel reduction of the new synchronized SP subgraph.

$$T = T \setminus SubF(U)$$

$$E_T = E_T \cup \{(h(U), b_U)\}$$

4 Example

An example of the algorithm applied to a given graph is shown, step by step, in Fig. 3,4,5. For each step, the first and second columns presents the graph and tree respectively, as a result of the previous step. For step 1 we present the original graph with a layering diagram and the root initialized tree. The third column is a diagram of the exploration phase on the tree. U and D node classes are shown with different grey shades, showing the graph related edges (not in the tree) by dashed lines. We also mark with names the orphan nodes, d-edges to further layers and the transitive, non-transitive over the U-handle property of the d-edges arriving at D classes. U-handles are marked with $h'(U)$ and final handles with $h(U)$. Forests under each handle are surrounded by trapezoids. New added synchronization nodes are represented with smaller circles.

We comment now the remarkable algorithm features in the example. Step 1 presents a case with only one U class with one node in the U class (handle) and two nodes in the induced D class. A new node 19 is added to the graph to synchronize over the nodes in the D class. In step 2, there are two U class to synchronize, being the handles the nodes in U classes. A d-edge appears from a node in the second class, and it source node 3 is changed in the original graph to the new synchronization node 21. Exploring phase in step 3, detects node 20 as the U-handle of the first U class as the nearest common ancestor of all nodes in U class (4,5). However, a d-edge to a node in the induced D class (21,11), which

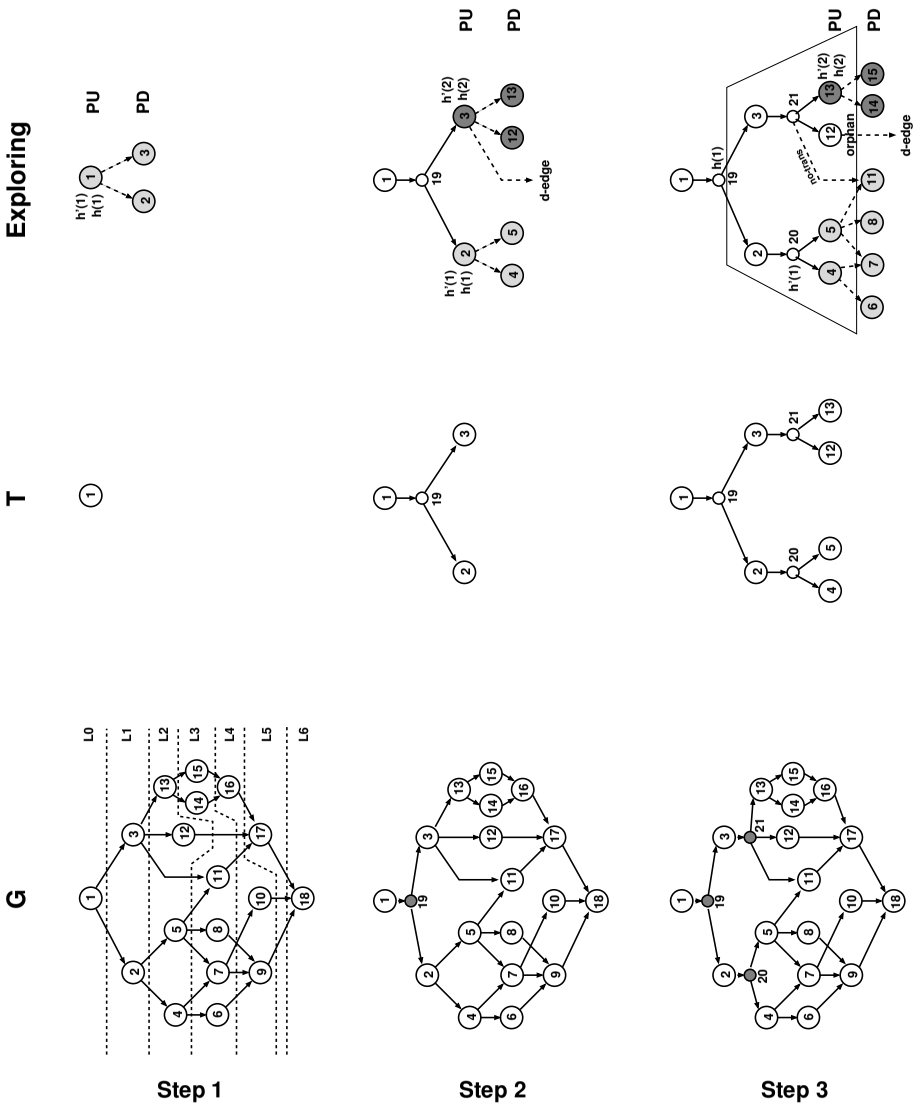

Fig. 3. Example of algorithm 2: Steps 1,2,3

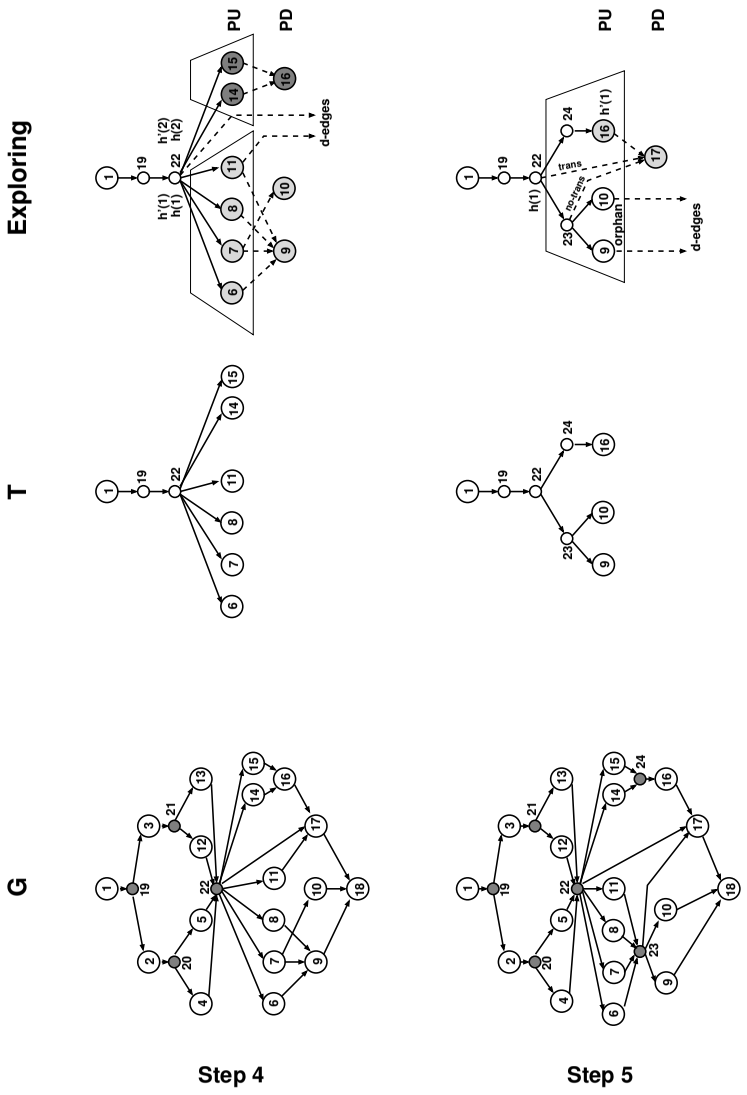

Fig. 4. Example of algorithm 2: Steps 4,5

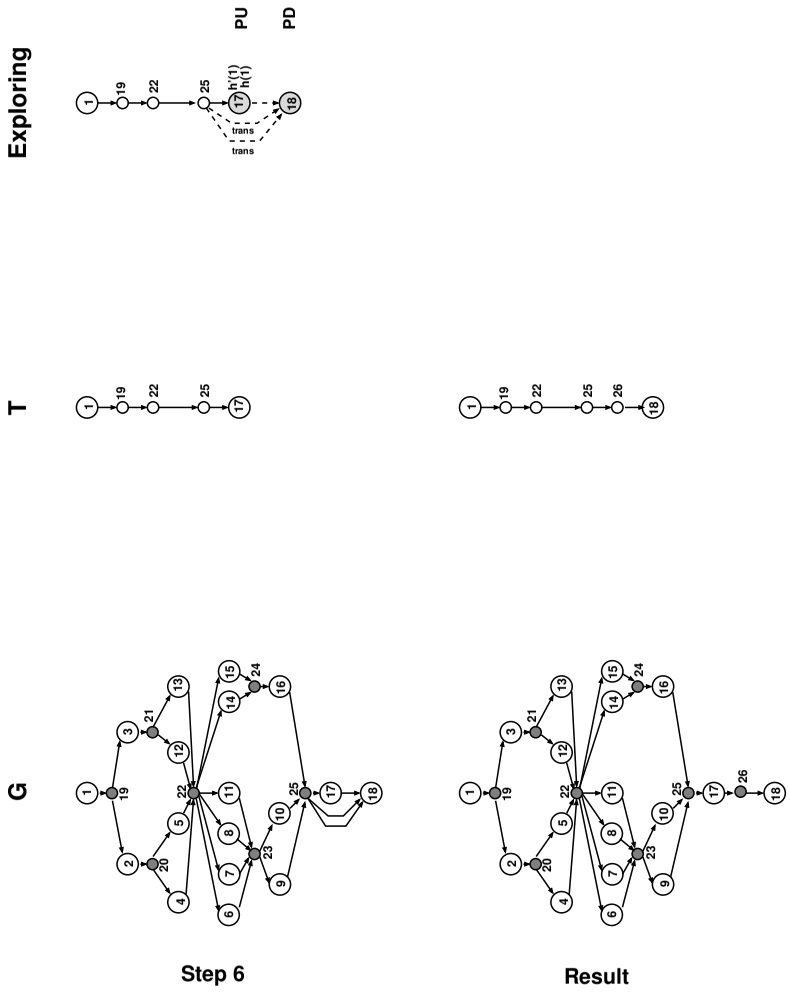

Fig. 5. Example of algorithm 2: Step 6 & Result

source node 21 is not transitive through the U-handle node 20, forces to explore further. The handle for class 1 is not equal to the U-handle, but the nearest common ancestor of nodes 20 and 21, namely node 19. Moreover, forests under the handles of classes 1 and 2 overlaps in node 13, and they are merged and synchronized together. Notice how the orphan node 12 is included in the merged U class and synchronized over the new node 22. Step 4 presents a situation where two U classes have the same handle node 22, but non-overlapping forests. Thus, they are not merged, but synchronized with different nodes 23 and 24. In step 5 there is only one U class, because nodes 9 and 10 have only d-edges to further layers. The U-handle is the same node 16 in U class. Nevertheless, there are d-edges from previous layers. Edge (22,17) is discarded due to its transitive property through the U-handle 16. However, edge (23,17) is not transitive. Thus, the handle node is the nearest common ancestor of nodes 16 and 23, namely node 22. The forest include now orphan nodes 9 and 10. In last step 6, there is only one U class and two discarded transitive edges. The resulting graph is shown together with the final tree, that is always a series graph in which each edge represents the minimal series-parallel reduction of a full SP subgraph.

5 Correctness

Since any tree can be easily transformed to a trivial SP StDAG, any graph which minimal series-parallel reduction is a tree, will be SP. As can be easily shown by induction on the depth of the StDAG, the minimal series-parallel reduction graph at each step is always a tree and, thus, has the SP property.

We define $TDAG$ as the subset of $DAGs$ that are one-rooted connected trees.

Proposition 1. *A tree is SP:*

$$T \in TDAG \Rightarrow T \in SP$$

Proof: The StDAG of T (called closure of T*) is the original T with an added leaf b_U connected to all the leaves $L(T)$. Applying series reduction to all originally leaves of T and parallel reductions where there were several leaves with the same parent, the result is equal to the closure StDAG of T', being T' the tree obtained eliminating $L(T)$ from T. Proceed recursively until only the root of T remains and the reduction is the trivial graph.*

Proposition 2. *A graph G which series-parallel reduction is a tree is SP.*

Proof: Compute the series-parallel reduction of G until it is a tree. As proved previously the series-parallel reduction of the closure of a tree is the trivial graph. Thus, the StDAG of the original graph can be series-parallel reduced to the trivial graph and is also SP.

5.1 Correctness proof:

1. **The result does not loose dependencies:** No node is eliminated from the graph. During synchronization, all times an edge (v, w) is eliminated it is substituted by two edges (v, b_U) and $b_U, w)$. Thus, the original dependence is transitively keep through b_U. All times a d-edge $(v, w) : v \in SubF(U), v \in L_j, j \leq i \wedge w \in L_k, k > i + 1$ is moved down to the synchronization node, the original edge dissapears and another edge (b_U, w) is added. After adding edges from U to b_U, $\forall u \in SubF(U), u \prec b_U$ and $v \prec b_U \prec w$.
   Thus, during the synchronization phase neither, the substitution of edges or moving down d-edges eliminate original dependencies in G. No other edge alteration is done in G.
2. **The result is SP:** We call S_i the subgraph of G that includes all nodes in layers $L_0, L_1, ..., L_i$ and all G edges incident to both nodes in S_i.
   When the algorithm begins (for $i = 0$) T is initialized with the root of G. S_0 is a one node tree. For $i = 1$ the closure of T and S_0 is computed and nodes in L_1 are hanged from the new synchronization node. T and S_1 are trees and, thus, they are SP.
   In each subsequent iteration (for $i = i + 1$), we compute $\mathcal{P}_U$, $\mathcal{P}_D$ and their handles. Then we merge classes with overlapping forests. Each forest is composed by trees that represent the series-parallel reduction of a subgraph of S_i. Eliminating in G edges from U to D and d-edges from $SubF(U)$ for all classes, S_i gets disconnected from the rest of the graph, being a tree (or a graph that is a tree after series-parallel reductions). New synchronization nodes and edges are added to closure every tree in T and G included in a forest of an U class. Thus, after synchronization, S_{i+1} is a tree or a graph that is a tree after series-parallel reductions. S_{i+1} is SP. T represents the series-parallel reduction of S_{i+1}.
   Proceed by induction until the last iteration. In last iteration (for $i = \hat{d}(G) - 1$), L_{i+1} is formed by the only one leaf of G. There is only one U class and one D class. All resting sub-trees in T (and G) are closed together with only one synchronization node and only one node (the leaf of G) is added hanging from that new node. T, that represents the series-parallel reduction of G is a series of nodes, its series reduction is the trivial graph. Thus, G is SP.

6 Critical path property for UTC graphs

An interesting feature of the algorithm is that it does not increase the critical path value if the original graph has unit time cost per node. Transforming a graph to SP form, this property minimizes the possibilities for critical path increment when no knowledge of the task load distribution is available.

Proposition 3. *For an UTC (Unit Time Cost) input graph G, the result G' is not UTC (nodes added by the algorithm carry no load), but despite the added dependencies, the critical path is not increased.*

Proof: For UTC graphs, the critical path value of G is equal to the maximum number of nodes that can be traversed from a root to a leaf ($cpv(G) = 1 + \hat{d}(G)$).

The algorithm adds zero loaded synchronization nodes between layers. The only way of increasing the critical path is due to added dependencies that make a node from a layer i dependent on a node from layer j, being $j > i$. However, the algorithm keeps the layers structure.

Moving d-edges sources to a node in a layer previous to the target node layer, does not change the depth level of any node. Substituting edges from nodes in U classes to nodes in D classes to include b_U nodes keeps the depth level of U nodes and adds one to the depth level of every node in D classes.

In the resulting graph, all even layers are populated by zero loaded nodes and odd layers by nodes in the original layers. The longest path from the root to the leaf alternatively crosses nodes with unit and zero time cost. The number of unit time cost nodes in the longest path is at most $1 + \hat{d}(G)$, and, thus, the critical path value in G' is the same as in G.

7 Complexity

7.1 Space complexity

Let n be the number of nodes and m the number of edges in the original graph. The number of nodes in the graph increases with one more node for each U class. Every node appears just once in an U class over the full algorithm run. Thus, the total number of nodes is upper bounded by $2n$. The number of edges is upper bounded because the processed subgraph (after each iteration) is SP, and the number of edges in an SP graph is bounded by $m \leq 2(n-2)$ (see proof in appendix A). Other ancillary structures (as the tree) store graph nodes and/or edges.

Thus, space complexity is:

$$O(m+n)$$

7.2 Time complexity

StDAG construction can be done in $O(n)$ and getting layers information in $O(m)$ with a simple graph search.

Classes of relatives for two consecutive layers can be computed testing a constant number of times each edge. Thus, all the classes along the algorithm run are computed in $O(m)$.

Exploration of the tree for handles can be self-destructive: Nodes are eliminated during the search. While searching for the handle of a class, all the forest can be eliminated and orphan nodes and other classes to be merged detected (see section 8 for a description of such an implementation).

Check and eliminate a transitive edge can be done in $O(1)$ if appropriate data structures are used for the tree [2], but assuming tree modifications are done in $O(\log n)$. $O(n)$ nodes and edges are inserted and eliminated in the tree. Thus, all tree manipulation has a time complexity $O(n \log n)$.

The synchronization phase adds $O(n)$ nodes, eliminate $O(m)$ edges and add a bounded number of edges ($O(n)$ because it is an SP graph). The movement of d-edges can be traced in $O(n \log n)$ with a tree-like groups joining structure to avoid real edge manipulation (see 8 for details).

Thus, time complexity is:

$$O(m + n \log n)$$

8 Implementation

We propose an implementation for the tree exploring phase. This implementation is based in a self-destructive search of the tree that eliminates the forests from the tree and detect handles with only one check per node. This implementation is needed to bound the time complexity as explained in section 7.

Searching for handles: For any given U class, we create an exploration structure call *explorers* (E). This structure stores nodes in sets indexed by depth level.

$$E = (m, V_E); m \in \mathbb{N}, V_E = \{V_1, V_2, ..., V_m\}$$

We initialize it with the nodes in any chosen U class.

$$\forall v \in U : V_{d(v)} = V_{d(v)} \cup \{v\}$$

$$m = \max d(v) : v \in U$$

For all nodes in E with maximum depth, we eliminate them from the tree, and we add the parent of the eliminated node to the explorers structure (avoiding repetition by marking the parent node when first visited).

To eliminate a tree node, we check previously if it is a leaf. If it is not, we proceed to eliminate all sub-trees hanging from it. The leaves of these sub-trees will be orphan nodes (that we immediately add to U) or nodes in other U classes. In this last both classes are merged, adding the new U nodes to the explorers structure.

When the explorers structure has only one node, this node is the U-handle $h'(U)$. Then we check the transitive condition of all d-edges arriving at D in the tree with $h'(U)$ to compute $K'_T(U)$. Non-transitive d-edges sources are added to explorers and the search is continued until the structure has again only one node. This last node is the handle $h(U)$, and is marked in the tree (a node can be handle of several classes at the same time).

During exploration, a node that is processed to be eliminated can also be marked as handle of other previously explored class or classes. In this case these classes are also merged with the one being explored.

When this exploring operation is performed for all U classes, all handles have been detected and marked, related classes already merged, and forests $SubF(U)$ deleted from the tree.

Tracking of d-edges: During the elimination of tree nodes we keep track of d-edges from these nodes to further layers. Each class maintains a set of these source nodes. When classes are merged, these sets are also merged. When a class is synchronized, this set will provide information for d-edges to be moved to the new synchronization node.

To keep track of d-edges movements without performing modifications in the graph, we use a tree-like set joining structure. The structure will map a node label to the node label of the final source of associated d-edges. A joining operation of a pair of node labels (i, j) will indicate that d-edges with source i are to be mapped to source node j. The structure has the property that for any sequence of joining operations $(i_1, j_1), (i_2, j_2), ..., (i_n, j_n)$ where $i_1 \neq i_2 \neq ... \neq i_n$ all joining operations take $O(n \log n)$ to be performed, and any mapping query takes $O(1)$.

Definition 1. *We define the* Joining structure $J = (\vec{I}, \vec{W}, \vec{S})$, *where* $\vec{I}, \vec{W}$ *are arrays of indexes and* $\vec{S}$ *is an array of sets of node labels (we define* N *as the set of all possible node labels). Let* $n = |V_G|$*:*

$$N = \{i : \mathbb{N}; i \in [1..2n]\}$$

$$\vec{I}, \vec{W} : N^{2n}$$

$$\vec{S} : S^{2n}, S_i \subseteq \{v : N\}$$

The J *structure is initialized as follows:*

$$I_i = i; W_i = i; S_i = \{i\}$$

It supports a joining operation indicating that i *must be mapped to* j *defined as:*

$$J \Uparrow (i, j) : J \to J'; J = (\vec{I}, \vec{W}, \vec{S}), J' = (\vec{I'}, \vec{W'}, \vec{S'});$$

$$I'_{(W_i)} = I_{(W_j)}$$

$$big = \begin{cases} W_i \ if \ |S_{W_i}| \geq |S_{W_j}| \\ W_j \ otherwise \end{cases}$$

$$small = \begin{cases} W_i \ if \ |S_{W_i}| < |S_{W_j}| \\ W_j \ otherwise \end{cases}$$

$$W'_i = W'_j = big$$

$$S'_{big} = S_{big} \cup S_{small}$$

$$\forall k \in S_{small} : W'_k = big$$

The query function is defined as:

$$J : V_G \to V_T; J(i) = W_i$$

9 Improvement: Non-necessary synchronization nodes

Some synchronization nodes can be eliminated. In situations where the U class, the induced D class, or both, have only one node, the new synchronization node is not necessary. The lonely node can play that role. This reduces the number of nodes and edges added, producing a completely equivalent graph result in terms of dependencies between nodes from the original graph.

We modify the algorithm synchronization phase along the following lines. For each final $U \to D$ classes:

- Detect/create synchronization node, and eliminate/add edges:
  1. If $U = \{u\}$, $b_U = u$:
     In G, eliminate all d-edges targeting a node in D.
     $$E_G = E_G \setminus \{(v, w) : w \in D, d(v) < i\}$$
  2. Else if $D = \{d\}$, $b_U = d$:
     In G, eliminate all d-edges targeting a node in D.
     $$E_G = E_G \setminus \{(v, w) : w \in D, d(v) < i\}$$
  3. Else (normal case where $|U| > 1, |D| > 1$): Proceed as in the original algorithm creating a new synchronization node b_U, eliminating in G all edges targeting a node in D, and adding edges from every node in U to b_U and from b_U to every node in D (barrier synchronization).
- Substitution of d-edges with source $v \in SubF(U)$, as in the original algorithm.
- Substitute the forest $SubF(U)$ in T for an edge $(h(U), b_U)$, as in the original algorithm.

In Fig. 6 we show the solutions obtained with the normal and improved algorithms for the same graph example used previously. The dependencies structure on the original graph nodes is the same, although the improved algorithm uses less nodes and edges.

10 Comparison with other algorithms

Other algorithms for the SP-ization problem used in our study are:

- Simple layering technique: Resynchronization by full barriers between each layer of nodes [5].
- Previous SP-ization algorithm introduced in [4].

Comparative characteristics discussed in this section are summarize in Table 1.

Layering technique is very simple and has a reduced time complexity ($O(n+m)$). It also maintains the layering structure of the original graph and has the property of not increasing the critical path value. However, it does not exploit the possibility of local resynchronization. The amount of dependencies added by

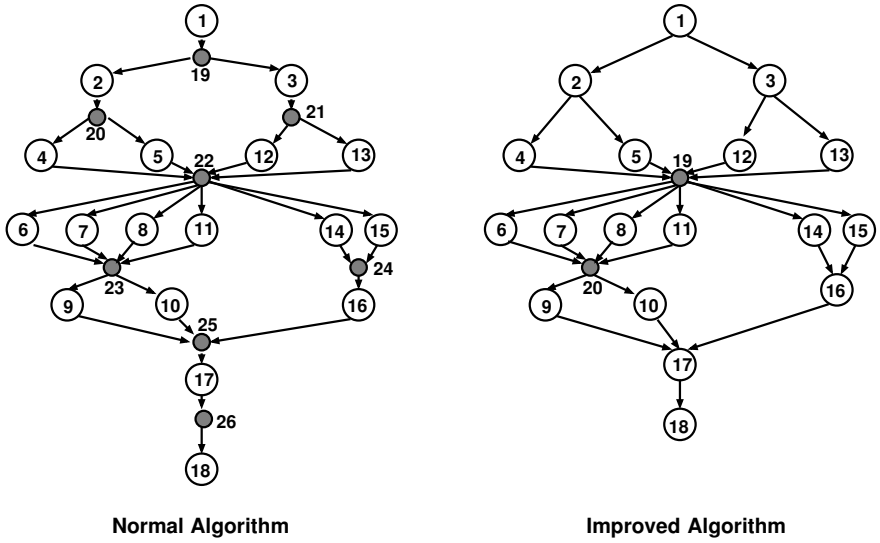

Fig. 6. Solutions obtained by the normal and improved algorithms

Algorithm	Space	Time	UTC-cpv	Regular graphs	Irregular graphs
Layering	$O(m+n)$	$O(m+n)$	Yes	Good	Bad
Previous SP-ization	$O(m+n)$	$O(m \times n)$	No	Good	Good
New algorithm	$O(m+n)$	$O(m+n\log n)$	Yes	Good	Good

Table 1. Algorithms comparison

the new algorithm is lesser or equal to the number of dependencies added by layering technique. For high edge density or high regular graphs, the solutions of both techniques are similar, if not the same. However, for low edge density or irregular graphs, the new algorithm finds highly improved solutions.

The previous SP-ization algorithm were based in iterative detection and resynchronization of local non-SP problems. The search for nodes implicated in such a problem was complex. Thus, time complexity bounds where much worse ($O(n \times m)$) and needed a transitive reduction of the original graph for high edge density graphs ($O(n^{2.81})$). The new algorithm uses the tree that represents the already SP processed subgraph to reduce the complexity order of searchs and transitive edges elimination. The previous algorithm uses a technique dependant on the searching order, thus, having the possibility of obtaining different solutions for the same graph topology depending on the node labeling. The new algorithm produce always the same solution for any input order of the same topology. The quality of the solutions is similar for both algorithms. However, the previous algorithm does not keep the layering structure in many situation, and may increase the critical path of UTC graphs due to added dependencies.

11 Conclusion

An efficient algorithm for the SP-ization problem is presented and analyzed in this report. The algorithm presents good features: A tight bounded complexity, local resynchronizations where possible and it maintains the layering structure of the graph, not increasing the critical path value for UTC graphs.

We conclude that this new algorithm improves solutions offered by previous algorithms studied (layering, and our first SP-ization technique), with tight complexity bounds.

Appendices

A Upper-bound of the number of edges in SP graphs

In this appendix we proof a result about SP graphs that we use to simplify the complexity measures.

Proposition 4. *Let $G = (V, E)$ be an SP multidigraph, and Gt a trivial graph. The parallel composition of G and Gt introduce a new edge in G from the root to the leaf (R_G, L_G) that is redundant (it was already in G) or is a transitive edge.*

Proof: If $(R_G, L_G) \in E$, the parallel composition with Gt introduces a redundant edge (R_G, L_G) by construction. If $(R_G, L_G) \notin E$, the parallel composition with Gt introduces the edge (R_G, L_G) by construction, which is a transitive edge because any $G \in SP$ is connected and thus, there exists a path $p(R_G, L_G)$ in G.

Proposition 5. *An SP digraph, different from the trivial graph Gt with no redundant and/or transitive edges cannot be derived from a parallel composition with a trivial graph Gt.*

Proof: From previous result. Any parallel composition with a trivial graph Gt, introduces a redundant or transitive edge.

Proposition 6. *Let $G = (V, E)$ be an SP multidigraph, different from the trivial graph Gt, with no redundant edges,*

$$\forall (v, w), (v', w') \in E; v \neq v' \vee w \neq w'$$

and no transitivities:

$$G^{-} = G$$

The cardinality of the edges set is bounded by:

$$|E| \leq 2(|V| - 2)$$

Proof: From previous result, an SP digraph with no redundant and/or transitive edges can be derived only from series compositions of any SP digraphs (included trivial graphs), and parallel compositions of non-trivial graphs.

1. *Let $G = (V, E)$ an SP digraph obtained by series composition of $k = 2, 3, ...$ trivial graphs Gt. The number of edges in G is by construction:*

$$|E| = |V| - 1 \leq 2(|V| - 2)$$

2. *Let $G = (V, E)$ an SP graph obtained by series composition of any SP digraph graph $G_1 = (V_1, E_1) : |E_1| \leq 2(|V_1| - 2)$ with a trivial graph Gt. By construction, the cardinality of V and E in G are:*

$$|V| = |V_1| + 1, |E| = |E_1| + 1$$

Thus,

$$|E| = |E_1| + 1 \leq 2(|V_1| - 2) + 1 < 2(|V_1| - 1) = 2(|V| - 2)$$

3. *Let* $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ *be two SP digraphs such that:*

$$|E_1| \leq 2(|V_1| - 2), |E_2| \leq 2(|V_2| - 2)$$

The series composition $G = (V, E)$*, by construction has the following number of nodes and edges:*

$$|V| = |V_1| + |V_2| - 1, |E| = |E_1| + |E_2|$$

Thus,

$$|E| = |E_1| + |E_2| \leq 2(|V_1| - 2) + 2(|V_2| - 2) = 2(|V_1| + |V_2| - 4) < 2(|V| - 2)$$

4. *Let* $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ *be two SP digraphs such that:*

$$|E_1| \leq 2(|V_1| - 2), |E_2| \leq 2(|V_2| - 2)$$

The parallel composition $G = (V, E)$*, by construction has the following number of nodes and edges:*

$$|V| = |V_1| + |V_2| - 2, |E| = |E_1| + |E_2|$$

Thus,

$$|E| = |E_1| + |E_2| \leq 2(|V_1| - 2) + 2(|V_2| - 2) = 2(|V_1| + |V_2| - 4) = 2(|V| - 2)$$

References

1. W. Bein, J. Kamburowski, and F.M. Stallman. Optimal reductions of two-terminal directed acyclic graphs. *SIAM Journal of Computing*, 6:1112–1129, 1992.
2. H. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, 1994.
3. R. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10, 303–318 1965.
4. A. González-Escribano, V. Cardeñoso, and A.J.C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st Euro-PDS Int'l Conf. on Parallel and Distributed Systems*, pages 251–256, Barcelona, July 1997.
5. A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, J. Alonso-López, D. Martín-García, and A. Pedrosa-Calvo. Measuring the performance impact of SP-restricted programming in shared-memory machines. In V. Hernández J.M.L.M. Palma, J. Dongarra, editor, *VECPAR 2000*, number 1981 in LNCS, pages 128–728, Porto (Portugal), June 2000. Springer.
6. J. Valdés, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *SIAM Journal of Computing*, 11(2):298–313, May 1982.