Clustering Algorithm for Scheduling Parallel Programs on NOWs with Synchronization Requirements at the Application Level

Bassel R. Arafeh
Department of Computer Science
Sultan Qaboos University
Muscat, Oman

Abstract

In this work, we are interested in developing an efficient heuristic algorithm for scheduling the tasks of a parallel program based on the class of UNC (Unbounded Number of Clusters) scheduling algorithms for clusters of NOWs. The main objective of the proposed UNC algorithm is to consider synchronous communication with deadlock avoidance strategy, for inter-task message-passing. The proposed algorithm generates non-linear clusters by traversing the task graph (DAG) once, using the Edge-Zeroing (EZ) technique. The objectives of the clustering algorithm is reducing the parallel time of the program, reducing the communication cost, improving the Program Computation to Communication Ratio (PCCR), and avoiding deadlock situations. The algorithm achieves its objectives with a time complexity $O(|V|(\log |V| + (|E|)^2))$ using nonlinear clustering in order to avoid more than one pass through the task DAG and to be able to deal with task DAG’s with fine, medium, and coarse granularity.

1. Introduction

With the advances in high-speed networks and processors hardware technology, parallel computing using a network of workstations (NOW) becomes reachable to a wide sector of users and application programmers with minimal extra cost. The number of interconnected computers by LAN, MAN, or WAN has increased tremendously in the last few years. For users, the clusters of NOW’s are powerful, cost-effective high performance computing systems that are underutilized most of the time. Typically, distributed-memory multicomputers have highly reliable interconnection networks for message-passing. However, communication over a network of workstations is not as reliable as that performed on parallel computer systems. Therefore, the requirement for synchronization at the application level to exchange information becomes eminent for many software systems, as in client/server applications. However, synchronous communication protocols may produce deadlocks between the communicating tasks. Accordingly, scheduling and mapping techniques for developing parallel programs on NOW’s must consider synchronous rather than asynchronous communication protocols when assessing the cost of inter-task communication, and be able to deal with deadlock situations with care.

In this work, we are interested in developing an efficient heuristic algorithm for scheduling the tasks of a parallel program based on the class of UNC (Unbounded Number of Clusters) scheduling algorithms for clusters of NOW’s. For an extensive overview on static scheduling techniques, algorithms, and tools see the survey article by Y. Kwok and I. Ahmad [1]. The main objective of the proposed UNC algorithm is to consider synchronous communication with deadlock avoidance strategy, for inter-task message-passing. That is, the algorithm should avoid merging a task to an already existing cluster, if it may cause a deadlock situation with some other cluster. The proposed algorithm generates non-linear clusters by traversing the task graph (DAG) once, using the Edge-Zeroing (EZ) technique. The objectives of the clustering algorithm is reducing the parallel time of the program, reducing the communication cost, improving the Program Computation to Communication Ratio (PCCR), and avoiding deadlock situations in the context of a synchronous communication environment. The algorithm achieves its objectives with a time complexity $O(|V|(\log |V| + (|E|)^2))$ using nonlinear clustering in order to avoid more than one pass through the task DAG and to be able to deal with task DAGs with fine, medium, and coarse granularity.

The remainder of the paper is organized as follows. In the next section, we introduce a background for the clustering problem, some related work, and the defini
tions of terms. In section 3, we describe the non-linear clustering algorithm for synchronous communication, called the NLC_SynchCom algorithm, its design principles, properties and complexity. In section 4, we present an example, and discuss the simulation and the experimental results. Finally, section 5 provides the conclusion.

2. Background

2.1 Preliminaries

In this work, a parallel program is modelled as a weighted directed acyclic graph (DAG), \( G = (V, E, \omega, \lambda) \), where \( V \) is the set of task nodes, \( E \) is the set of communication edges, \( \omega \) is the set of task computation weights, and \( \lambda \) is the set of edge communication costs. An edge \( e_{ij} = (n_i, n_j) \in E \) represents a data dependence constraint between the two tasks \( n_i \) and \( n_j \), where the execution of \( n_j \) must start after receiving all input from \( n_i \). The communication cost of message passing along an edge \( e_{ij} \) is denoted by \( c_{ij} = \lambda(e_{ij}) \), and the computation weight of a task \( n_i \) is denoted by \( \omega(n_i) \). We will refer to the source node and destination node of an edge as the parent node and the child node, respectively. A node that does not have any parent is called an entry node; while a node which does not have any child is called an exit node. \( \text{Pred}(n_i) \) is the set of immediate predecessors of \( n_i \), and \( \text{Succ}(n_i) \) is the set of immediate successors of \( n_i \). A task node with two or more children is called a fork task. On the other hand, a task node with two or more parents is called a join task. The computation to communication ratio of a parallel program (PCCR) is defined as its average computation weight divided by its average communication cost.

The execution behavior of the program DAG is the macro-dataflow model. However, the execution of each task consists of three phases: Receive, compute, and send. The receive phase includes receiving all messages required by the task for its execution to start. Messages may arrive overlapped in time and handled instantaneously. The compute phase is the phase in which the instructions of the task are executed without interruption. We assume a synchronous communication protocol. The send phase includes sending all messages to all dependent tasks in parallel. However, the sender task is blocked until all task modules actually receive the messages.

2.2 Clustering Problem

A clustering of a DAG, \( G \), is a mapping of the nodes in \( V \) onto clusters, where each cluster is a subset of \( V \) that is assigned to a virtual processor. The virtual architecture is a distributed multicomputers environment with unbounded number of processors that are completely connected. A DAG with a given clustering is called a clustered DAG. A schedule for a clustered DAG assigns a task execution order to each task node \( n_i \) on each processor \( p \) it is mapped to, and it is called a scheduled DAG. All tasks in a cluster must execute in the same processor. The communication cost of an edge in a clustered DAG becomes zero if the source and destination nodes of this edge are in the same cluster. Two approaches for clustering are used: linear and nonlinear. A cluster is called linear if tasks that are in a precedence path of the DAG are allocated to the same cluster. A cluster is called nonlinear if two independent tasks are allocated to the same cluster. A linear clustering exploits the potential parallelism in the DAG, while a nonlinear clustering reduces parallelism by sequentializing parallel tasks.

Clustering may be used as the first step in a two-step compile-time procedure for scheduling the tasks graph on parallel architectures, as proposed by Sarkar [2]. The second step would include merging the clusters onto the number of available physical processors and scheduling their execution. A clustering technique has a set of objectives to achieve. Mainly, the objectives are: 1) reduction of parallel time on an unbounded number of processors, 2) reduction of the communication cost, and 3) maximization of the efficiency.

A critical path of a DAG is the path from an entry node to an exit node, of which the sum of computation weights and communication costs is the maximum. Indeed the parallel time of a program DAG is determined by its critical path. However, the parallel time of a clustered DAG is determined by the critical path of its scheduled DAG rather than by its clustered DAG. It is called by Yang and Gerasoulis [3] the dominant sequence to distinguish it from the critical path of a clustered DAG.

The clustering problem has been shown to be NP-complete, with or without node duplication [4] [5] [2]. For clustering techniques without node duplication, polynomial-time heuristic algorithms have been proposed based on the critical path analysis by Sarkar [2], Wu and Gajski [6], Gerlasoulis and Yang [7] [3], Kwok and Ahmad [8], and Kadamuddi and Tsai [9].

The message-passing system for communication between processors has been assumed to be asynchronous in most proposed algorithms for clustering. In a syn-
Boolean Merge_Child(parent, child)
If (child status is Head of its cluster or a Singleton) and
(parent status is Tail of its cluster or a Singleton)
Find the end send of parent, S(parent), in case child is merged with
Clust(parent);
If S(parent) > e_receive(child) return(False)
Else
Clust(parent) = Clust(parent) +{child};
Update the status of parent and child nodes in Clust(parent);
Apply the transitive closure rule on all successor nodes of parent i
Clust(child), if any, except the child node;
Return(True);
EndIf
Else
If (parent status is a Singleton) compute the end receive of parent,
R(parent), in case of merging and linearization of the parent with
the Clust(child);
If (R(parent) > e_receive(parent) return(False);
Compute the end receive of child, R(child), in case of merging and
linearization of the parent with the Clust(child);
If R(child) > e_receive(child) return(False)
Else
Clust(child) = Clust(child) +{parent};
Update the status of parent and child nodes in Clust(child);
Apply the linearization procedure on Clust(child);
Return(True);
EndIf
EndIf
End Merge_Child

Figure 1: The Merge_Child algorithm for merging and linearization.

chronous communication the sender is blocked until an acknowledgment is received from the receiver. This waiting time is called the blocking delay. Accordingly, a clustering algorithm may generate a deadlock situation because of the use of synchronous communication protocol. A deadlock occurs when a sender gets blocked indefinitely waiting for an acknowledgment from a receiving task in another cluster; while the receiver cannot start execution because of a predecessor task in the same cluster has been blocked indefinitely for the same reason. Based on the task execution phases, the following time parameters characterize the schedule of a task node in a scheduled DAG:

- $s_{receive}(n_i)$: start time for receiving messages by task node $n_i$.
- $e_{receive}(n_i)$: end time for receiving messages by task node $n_i$.
- $s_{compute}(n_i)$: start time for computation by task node $n_i$.
- $e_{compute}(n_i)$: end time for computation by task node $n_i$.

3. Nonlinear Clustering Algorithm for Synchronous Communication

3.1 Merging and Linearization

There are two forms for merging a parent node with one of its child nodes so that they can be in the same cluster. The selection of which form to be used is based on the status of the parent and the status of the child. In the first form, the parent node must be a Singleton or the Tail of its cluster, and the child should be a Singleton or the Head of its cluster. In the second form, the parent node must be a Singleton, and the child node should be the Tail or a Regular node of its cluster. The first form of merging is an extended form of linear clustering, since a child node could be a join node, referred to as extended-linear. While the second form of merging is a restricted non-linear clustering, since a child is definitely a join node, referred to as restricted nonlinear.

The tasks of each non-linear cluster must be scheduled for execution on the same processor. Therefore, a schedule enforces an ordering for the execution of tasks, which makes them considered as linear clusters in a scheduled DAG. We call the process of converting a nonlinear cluster to a scheduled one (i.e., a linear cluster) by Linearization. A cluster is linearized after

Linearize(parent, child)
Apply the transitive closure rule on all predecessor nodes of the parent in Clust(child);
Apply the transitive closure rule on all successor nodes of the parent in Clust(child);
Find the immediate predecessor node, u, of the child node in Clust(child);
Create a virtual edge between u and parent, and
deactivate the edge (u, child);
Modify the time parameters for the parent and child nodes in the table of time parameters.

Figure 2: Linearize algorithm for linearization.

- $s_send(n_i)$: start time for sending messages by task node $n_i$.
- $e_send(n_i)$: end time for sending messages by task node $n_i$.  

0-7695-1573-8/02/$17.00 (C) 2002 IEEE
it has gone through merging a node to it. The type of linearization applied depends on the form of merging performed. The algorithm for merging and linearization is referred to by the Merge Child algorithm, and it is shown in Figure 1.

For the extended-linear merging, the cluster of the child node is merged with the cluster of the parent node, and the status of both nodes are changed accordingly. The clusters of both, the parent and child nodes engaged in a merging step, are assumed to be in a linearized form before merging is applied. However, the extended-linear form for merging necessitates to have a linearized transition from the parent’s cluster to the child’s cluster. This linearization is ensured by applying the transitive closure rule. That is, the path from a predecessor node \( v_i \in \text{Pred}(\text{child}) \), where \( v_i \) is in the \( \text{Clust}(\text{parent}) \), to the child node has to go through the \((\text{parent}, \text{child})\) edge. Effectively, the application of the rule would mark all such edges from a parent’s cluster to the child node as Inactive, except the \((\text{parent}, \text{child})\) one. Similarly, the transitive closure rule is applied on all successors of the parent node in the child’s cluster. The time complexity of applying the transitive closure rule is \( O(|E|) \).

For the restricted-nonlinear merging, the Singleton parent node is merged with the child’s cluster, and the status of both nodes are changed accordingly. In this case, a linearization step is performed to schedule the execution of the parent task in the child’s cluster. In general, the major concern for zeroing an edge at a merging step \( j \) is that it does not cause an increase in the DAGs parallel time, \( PT \). That is, \( PT_j \leq PT_{j-1} \). This constraint is implicitly satisfied in the case of the extended-linear merging according to the definitions of the \( e_{\text{receive}} \) and \( e_{\text{send}} \) time parameters.

However, the constraint must be enforced explicitly when the restricted-nonlinear merging form is applied, as shown in the Merge Child() algorithm. In this case, the parallel time, \( PT_j \), of a DAG may increase after a merging step \( j \) because of a linearization step. Obviously, the execution order of the parent task in the child’s cluster may have an effect on its \( s_{\text{compute}} \) time and the \( s_{\text{compute}} \) of the child task. We will extend the notations for the time parameters here to include the merging step, \( j \), at which the computation is performed, for example \( e_{\text{receive}}(\text{parent}, j) \). The constraint for merging and linearization in the case of the restricted-nonlinear form is characterized by the following conditions:

\[
\begin{align*}
e_{\text{receive}}(\text{parent}, j) & \leq e_{\text{receive}}(\text{parent}, j - 1) \\
e_{\text{receive}}(\text{child}, j) & \leq e_{\text{receive}}(\text{child}, j - 1)
\end{align*}
\]

Algorithm: NLC_SynchCom Algorithm

1. Initialization phase to determine for each node \( n \) the following sets: \( \text{Pred}(n), \text{Succ}(n), \text{Level}(n), \text{Clust}(n), \text{etime}(n) \);
2. Construct the table of time parameters and initialize its entries for each node \( n \) by: \( s_{\text{receive}}(n), e_{\text{receive}}(n), s_{\text{compute}}(n), e_{\text{compute}}(n), s_{\text{send}}(n), e_{\text{send}}(n) \);
3. While there is an unexamined level \( l \) Do
4. Sort the nodes at level \( l \) in descending order of their end of sending times (\( e_{\text{send}} \));
5. For each node \( n_1 \) at level \( l \) Do
6. If \((\text{\textbf{n1} is a Singleton or the Tail of the Clust(n1)) and Succ(n1) is not empty}) Then
7. While there is an unexamined child \( n_2 \) in Succ(n1) Do
8. Select a child \( n_2 \) in Succ(n1);
9. For each node \( p \) in Pred(n2), such that \( p \) is not equal to \( n_1 \);
10. Apply the Deadlock detection function between Clust(n1) and Clust(p) to detect the existence of a deadlock situation assuming \( n_2 \) to be merged with \( n_1 \);
11. If no deadlock is found between Clust(n1) and Clust(p) Then
12. Apply the Merge Child(n1, n2) procedure;
13. If the merging is successful Then break;
14. Update the time parameters in the DAG affected by the previous merging step.
EndWhile
EndIf
EndFor
EndWhile

Figure 3: NLC_SynchCom algorithm for synchronous communication.

The linearization algorithm is described in Figure 2, and the assessment of its complexity is shown through Lemma 1.

Lemma 1 The time complexity of a linearization step for a restricted-nonlinear merging is \( O(|E|) \).

3.2 Description of the Algorithm

In this section, we present a clustering algorithm for parallel applications with synchronization requirements at the application level. The algorithm proceeds in one pass in the forward direction from entry nodes to exit nodes, one level at a time. The algorithm starts assuming each task node is in a cluster by itself. Therefore, there are \( |V| \) clusters at the beginning of the algorithm. Each node in a cluster is designated by its status as a Head, Tail, Regular or Singleton. A node in a cluster by itself is given the status of a Singleton
node. The other node designations are clear from their names. The selection of a parent node, \( n_i \), at level \( l \) that will be merged with one of its child nodes is determined using two priority schemes. The first scheme is used to determine the priority of a parent node at level \( l \) for merging. It is defined by the parent’s completion time, \( c_{send}(parent) \), in descending order. The second scheme is used to determine the priority of merging a child node \( n_j \) for a parent \( n_i \). This priority depends on the maximum remaining time left to the completion of execution from a parent node, \( n_i \), to an exit node, excluding \( n_i \), and it is given by

\[
\max_{n_k \in \text{Successors}(n_i)} \{ c_{receive}(n_k) - s_{send}(n_i) + \text{etime}(n_k) \}
\]

where \( \text{etime}(n_k) \) refers to the execution time remaining starting from \( n_k \) to an exit node, called the exit time, and it is defined as follows.

**Definition 1** The exit time of a node \( n_k \) is determined as follows

\[
\text{etime}(n_k) = \begin{cases} 
\omega(n_k), & \text{if } n_k \text{ is an exit node} \\
\omega(n_k) + \max_{n_j \in \text{Successors}(n_k)} \{ c_{receive}(n_j) - (s_{send}(n_k) + \text{etime}(n_j)) \} & \text{otherwise}
\end{cases}
\]

Since a merging step would zero a (parent, child) edge, then the completion time of the parent node and all its descendant nodes may have their completion times changed accordingly. Thus, the priorities of the parent nodes at each level are found dynamically before the nodes of that level are scanned for merging. On the other hand, the exit time for each node is computed at the initialization time only. Deadlock detection is performed incrementally, whenever a child node is selected for merging. In the following paragraphs, we address the issue of deadlocks in some details. First, the deadlock situation is characterized by Lemma 2. Second, the time complexity of detecting the existence of a deadlock situation between clusters is shown in Corollary 1. We will refer to the execution order of two tasks \( v_1, v_2 \), where \( v_1 \) is a predecessor of \( v_2 \) in a linear cluster, by \( v_1 \ll v_2 \) to mean that \( v_1 \) is executed before \( v_2 \).

**Lemma 2** Given two independent linear clusters, \( \text{Clust}_1 \) and \( \text{Clust}_2 \) in a clustered DAG, \( G = (V, E, \omega, \lambda) \), we say that \( \text{Clust}_1 \) and \( \text{Clust}_2 \) are in deadlock if there are at least four task nodes, say, \( v_1, v_2 \in \text{Clust}_1 \) and \( v_3, v_4 \in \text{Clust}_2 \), such that \( v_1 \ll v_2; v_3 \ll v_4 \) and the edges \( (v_1, v_4), (v_3, v_2) \in E \).

**Corollary 1** The time complexity to determine the existence of a deadlock situation between two independent linear clusters, \( \text{Clust}_1 \) and \( \text{Clust}_2 \), due to a join task node, say \( v_1 \in \text{Clust}_1 \), is \( O(|E|) \), if there is a parent node of \( v_1 \), say \( v_2 \), such that \( v_2 \in \text{Clust}_2 \).

The description of the nonlinear clustering algorithm is presented next. The non-linear clustering algorithm for synchronous communication is shown in Figure 3, called NLC\(_{\text{SynchCom}}\). The algorithm starts with an initialization phase in step 1, which determines the set of predecessors, \( \text{Pred}(n) \), set of successors, \( \text{Succ}(n) \), node level, \( \text{Level}(n) \), and cluster number, \( \text{Clust}(n) \), for each node \( n \in V \). The nodes in the \( \text{Succ}(n) \) set are sorted in descending order based on their exit times. In case of a tie, the tied nodes are ordered based on the edge communication cost. Initially each task node is in a cluster by itself. In step 2, A table of time parameters is constructed and initialized by the time parameters for each node as initially given. The table of time parameters will be used frequently during the clustering algorithm; it is updated incrementally as a children edge is zeroed by a merging step.

The algorithm scans the nodes of the DAG top to bottom level, as indicated in step 3. Step 4 sorts the nodes at level \( l \) in descending order of \( c_{send} \) times. At each level \( l \), nodes are visited in order as indicated by the for-loop in step 5. A Parent node \( n_1 \) at level \( l \) can be merged with one of its child nodes, if its \( \text{Succ}(n_1) \) set is not empty, and if it is a Singleton or the Tail of the \( \text{Clust}(n_1) \). This check is done by step 6 in the algorithm. If the conditions are not satisfied, the next node at level \( l \) is selected. All children of the selected parent, \( n_1 \), are candidates for merging with the parent node, as indicated by the while-loop in step 7, each according to its priority level. In step 8, a child node \( n_2 \) is selected from the top list of \( \text{Succ}(n_1) \). A priority is given to merge the parent node with a child which has the highest exit time. Deadlock detection is performed in steps 9 and 10. All predecessor nodes, \( p \), of the candidate child are scanned, except the parent node \( n_1 \), to check whether \( \text{Clust}(n_1) \) and \( \text{Clust}(p) \) might generate a deadlock, if merging between the parent and child nodes is performed. If no deadlock is found, the selected child node, \( n_2 \), has to pass another test before merging can be finally applied, as shown in steps 11, 12 and 13. A new child node is selected from the set \( \text{Succ}(n_1) \); if there has been a deadlock detected due to the current candidate child, or the current merging step is not successful. In step 14, the algorithm updates the table of time parameters of the DAG affected by the merging steps at level \( l \).

The complexity of the NLC\(_{\text{SynchCom}}\) algorithm is assessed by the following theorem.
Table 1: Exit times for the DAG Nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>etime</td>
<td>33</td>
<td>33</td>
<td>26</td>
<td>23</td>
<td>26</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

**Theorem 1** The time complexity of the NLC_SynchCom algorithm for synchronous communication with deadlock avoidance is $O(|V| \log |V| + (|E|)^2)$.

**Proof.** The proof is based on the previous discussion and omitted for the lack of space. ■

Figure 4: The weighted DAG of the example

Figure 5: The Gantt chart for the scheduled DAG without clustering.

**4. Simulation and Experimental Results**

**4.1 Example**

The task DAG shown in Figure 4 is used to illustrate the NLC_SynchCom algorithm. We use a Gantt chart to demonstrate the scheduling of the DAG task nodes as determined by the entries of the table of time parameters: $s_{receive}$, $e_{receive}$, $s_{compute}$, $e_{compute}$, $s_{send}$, $e_{send}$. The Gantt chart in Figure 5 shows the scheduling of the DAG task nodes before applying the NLC_SynchCom algorithm for clustering. The computation of the exit times, $etime(n_i)$, are shown in Table 1. Initially, the $PT$ is 33 units, the total communication cost is 53 units and the PCCR is 0.68. The final results after clustering are shown in Figures 6 and 7. The scheduled DAG of the example has a $PT$ of 25 units after clustering, with a PT reduction of 24.2% from the initial value. The communication cost has been reduced by 60.4% and the PCCR after clustering becomes 1.71.
4.2 Simulation

We have implemented the NLC_SynchCom algorithm for nonlinear clustering for simulation and testing, using randomly generated DAGs. Random DAGs are generated by determining the number of levels in each DAG randomly. Then, we determine the number of tasks in each level based on a randomly generated width at that level. The out-degree of each created node is determined randomly. Accordingly, we generate edges from each node at level \( l_1 \) to a randomly selected node at a level \( l_j \), where \( j > i \). The computation weight for each node is randomly assigned. The communication cost of each edge is determined randomly based on the selected granularity for the DAG to be generated and the average computation weight of the DAG.

The following information are used for generating random weighted DAGs and analyzing the statistical data generated by the simulation results: Level range, width range, maximum computation weight, maximum out-degree, and the granularity range. We are interested in testing the effectiveness of the NLC_SynchCom algorithm heuristic for nonlinear clustering using different DAG sizes and different granularity. Seven groups of DAGs are generated. Each group is identified by its level range and width range. The groups of random DAGs generated are shown in Table 2, they are referred to by G1-G7. Three categories of weighted DAGs are tested. Each category is associated with a range of the ratio of average computation weight over the edge communication cost, which approximates the graph granularity. It is referred to here as the CCR range. The weighted DAG categories are:

- Fine Grain: The CCR range is selected to be 0.2-0.8.
- Medium Grain: The CCR range is selected to be 0.8-1.4. In this case the average computation weight is close to the average communication cost.
- Coarse Grain: The CCR range is selected to be 2.0-8.0.

Under each category, we have generated 100 random DAGs for each level and width range, in order to simulate different sizes of graphs. We have kept the maximum computation weight and the maximum out-degree the same over all simulation runs; and these are 20 and 5, respectively. The average edge communication cost before clustering is the same over all runs for the same CCR range. For each category, we have measured the average edge communication cost after clustering, the PCCR before and after clustering, the PT before and after clustering, and the percentage number of DAGs with one or more deadlocks in that run. The simulation results for the three DAG categories are shown in Tables 3, 4, and 5.

The average edge communication costs before clustering for categories 1, 2, and 3 are 29.94, 9.9, and 3.5, respectively. The average reduction in communication cost due to clustering is about 0.47, 0.45, and 0.45 for categories 1, 2, and 3, respectively. The average improvement in PCCR after clustering has been 93%, 84%, and 88% for categories 1, 2, and 3, respectively.

We recognize that the NLC_SynchCom algorithm is most effective for fine grain DAGs. It achieves higher reduction in the parallel time and higher percentage of improvement in PCCR for comparable DAG sizes.

| Table 2: The groups of DAG’s randomly generated |
|---------------------|---------|---------|---------|
| Group    | Level Range | Width Range | \( |V|_{ave} \) | \( |E|_{ave} \) |
| G1       | 3-7         | 3-5       | 15.88    | 13.81    |
| G2       | 3-7         | 7-10      | 35.0     | 31.65    |
| G3       | 8-12        | 3-5       | 33.33    | 35.1     |
| G4       | 8-12        | 8-12      | 90.39    | 99.0     |
| G5       | 8-12        | 13-16     | 134.36   | 145.7    |
| G6       | 16-20       | 8-12      | 167.1    | 191.97   |
| G7       | 16-20       | 16-20     | 304      | 348.91   |

| Table 3: Simulation results for fine grain DAG’s |
|---------------------|---------|---------|---------|---------|---------|---------|
| Group    | \( \lambda_{ave} \) after | PCCR before | PCCR after | PT before | PT after | PT red. % |
| G1       | 13.5     | 0.44     | 1.14     | 119.3     | 12.5     |
| G2       | 14.61    | 0.4      | 0.82     | 139.0     | 17.3     |
| G3       | 15.71    | 0.33     | 0.63     | 188.74    | 16.6     |
| G4       | 16.52    | 0.31     | 0.56     | 222.41    | 13.6     |
| G5       | 16.85    | 0.31     | 0.55     | 237.86    | 15.3     |
| G6       | 17.08    | 0.3      | 0.51     | 292.96    | 12.2     |
| G7       | 17.12    | 0.3      | 0.51     | 311.08    | 11.1     |

| Table 4: Simulation results for medium grain DAG’s |
|---------------------|---------|---------|---------|---------|---------|---------|
| Group    | \( \lambda_{ave} \) after | PCCR before | PCCR after | PT before | PT after | PT red. % |
| G1       | 4.79     | 1.32     | 3.02     | 65.32     | 11.3     |
| G2       | 5.09     | 1.2      | 2.37     | 75.33     | 9.9      |
| G3       | 5.38     | 1.01     | 1.83     | 97.22     | 9.4      |
| G4       | 5.61     | 0.94     | 1.65     | 118.32    | 8.2      |
| G5       | 5.73     | 0.95     | 1.63     | 122.28    | 8.3      |
| G6       | 5.78     | 0.89     | 1.52     | 150.39    | 7.0      |
| G7       | 5.81     | 0.89     | 1.51     | 160.52    | 6.5      |
Table 5: Simulation results for coarse grain DAG’s

<table>
<thead>
<tr>
<th>Group</th>
<th>$\lambda_{ave}$ after</th>
<th>$PCCR$ before</th>
<th>$PCCR$ after</th>
<th>$PT$ before</th>
<th>$PT$ red. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>1.66</td>
<td>3.74</td>
<td>9.19</td>
<td>49.47</td>
<td>5.7</td>
</tr>
<tr>
<td>G2</td>
<td>1.82</td>
<td>3.38</td>
<td>6.68</td>
<td>57.68</td>
<td>5.7</td>
</tr>
<tr>
<td>G3</td>
<td>1.92</td>
<td>2.82</td>
<td>5.18</td>
<td>72.55</td>
<td>5.5</td>
</tr>
<tr>
<td>G4</td>
<td>1.98</td>
<td>2.67</td>
<td>4.71</td>
<td>86.72</td>
<td>4.7</td>
</tr>
<tr>
<td>G5</td>
<td>2.00</td>
<td>2.69</td>
<td>4.67</td>
<td>90.74</td>
<td>4.6</td>
</tr>
<tr>
<td>G6</td>
<td>2.02</td>
<td>2.55</td>
<td>4.38</td>
<td>110.07</td>
<td>4.1</td>
</tr>
<tr>
<td>G7</td>
<td>2.03</td>
<td>2.54</td>
<td>4.32</td>
<td>117.27</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Furthermore, the amount of $PT$ reduction and $PCCR$ improvement decreases as the size of the DAGs grow in the same category. It has been recorded that, on average, one third of the DAGs randomly generated have one or more deadlocks detected.

5. Conclusion

In this paper, we have presented an efficient non-linear clustering algorithm for scheduling tasks on unbounded number of processors, assuming inter-task synchronous communication. We have proven the effectiveness of the algorithm in preventing performance degradation due to synchronous communication and deadlock avoidance using non-linear clustering approach. Simulation results have shown the scalability of the algorithm and the extent of its performance in improving $PCCR$, and reducing the communication cost and the parallel time. The contribution of the work is towards facilitating parallel computing on clusters of NOW’s. however, the proposed algorithm can be used on any distributed-memory architecture. For applicability to practical environments, a scheme is needed to extend the algorithm to handle scheduling tasks on bounded number of processors (BNP) with arbitrary interconnection networks.

References


