2018 IEEE International Conference on Computational Science and Engineering

A time-triggered scheduling algorithm for active diagnosis in heterogeneous distributed systems

Sarah Amin
Embedded Systems Lab
University of Siegen
Siegen, Germany 57076
Email: sarah.amin@uni-siegen.de

Roman Obermaisser
Embedded Systems Lab
University of Siegen
Siegen, Germany 57076
Email: roman.obermaisser@uni-siegen.de

***Abstract*—Many safety-critical control applications have now started using distributed embedded systems that consist of sensors, actuators, processors and a fully connected network with no mechanical backup. One such example is a fly-by-wire control system used in modern day aircraft. These systems require a high level of reliability, safety and performance. Many redundant components are added to ensure the reliability of such systems but this is not a viable solution because it increases the overall size of the system along with its cost and power consumption. A better approach is to continuously monitor the system to ensure that the dependability of the overall system is greater than that of its hardware and software components. Active diagnosis is one such technique that monitors the operation of the system components at runtime for fault isolation and error recovery. Since such systems are highly safety-critical i.e. their failure can result in loss of life etc., therefore it is essential that the analysis of the diagnostic information and subsequent recovery from the fault is performed within predictable time. Since scheduling the diagnostic tasks on to the system is an important aspect of this timely analysis, so the present work proposes a time-triggered static scheduler for a heterogeneous distributed system that uses diagnostic queries and a real-time database to find faults. The proposed algorithm calculates the points in time at which each diagnostic query is executed or data is replicated to or deleted from the real-time database. This a priori knowledge bounds the time for fault identification and results in a realizable diagnostic framework. The algorithm utilizes a specific priority scheme to incorporate the heterogeneity of the system and schedules the diagnostic tasks while respecting their precedence and periodicity constraints. It is prototypically implemented and is experimentally evaluated. The paper demonstrates that time-triggered scheduling can be successfully incorporated in a multi-query based diagnostic environment to increase the reliability, performance as well as to ensure the safety of a heterogeneous distributed system.**

***Index Terms*—distributed fault analysis, diagnosis, active diagnosis, real-time, distributed systems, list scheduling, time-triggered scheduling, heterogeneous systems, diagnostic queries**

I. INTRODUCTION

In recent years, many control applications, for example automotive systems [1], electrical power distribution systems [2], command/control systems [3], networked medical devices and health management systems, nuclear systems [4], railway systems [5] etc., have started using distributed embedded systems that consist of sensors, actuators and a fully connected network with no mechanical backup. For example, a modern pacemaker is a computer based device with specialized peripherals, nuclear power plants use computerized systems to shut down the plant in case of a safety hazard [4], modern aircrafts use fly-by-wire (FBW) control system that relies heavily on computer networks [6] and an embedded system based steer-by-wire (SBW) control system that provides steering controls to modern day cars [5]. Since such systems are highly safety-critical i.e. their failure might result in loss of life, damage of property or damage of environment so it is essential that they are equipped with a fault-tolerant system [7]. There are two solutions in literature to ensure the reliability of such systems. Firstly, redundant components can be included in the end product but it is not viable as these systems have severe cost constraints [7]. Another solution is to continuously monitor the system for permanent or temporary component failures. There are two types of diagnostic techniques in the literature, i). passive diagnosis that stores the diagnostic information and analyzes it later for maintenance and feedback and ii). active diagnosis that analyzes the diagnostic information at runtime so that an immediate recovery action can be taken.

In the state-of-art, a lot of work has been done in the field of fault diagnosis to improve the reliability, safety and performance of distributed systems. One of the earliest diagnostic technique used the PMC model [8] to identify permanent failures in a distributed embedded system. This technique was later modified by authors in [9] to identify temporary failures in the system. Both these models work on the assumption that processors can test each other and compare results to identify underlying faults. However, it is not a viable approach as it is extremely hard to acquire test results. To overcome this problem, many comparison-based methods [10]–[15] were introduced. In these methods, tasks are duplicated on different processing elements and then their end results are compared to identify the faulty components in the system. Similar model-based techniques were proposed by authors in [16]–[22] where the system behavior was compared with a reference model to identify faults.

In safety-critical systems, it is important that a fault is determined and the problem is solved within predictable time before the system reaches an unsafe state. Most control applications manage the loss of control inputs only for a few cycles but longer outages shut off the system completely due

978-1-5386-7649-3/18/$31.00 ©2018 IEEE
DOI 10.1109/CSE.2018.00013

to the incurring fault. For example, in a steer-by-wire the maximum time for which the control systems can freeze the actuators is 50ms [23]. This aspect raises the need of a diagnostic solution that respects the timing constraints in control applications and identifies faults within predictable time. Most of the aforementioned techniques do not consider stringent timing constraints that are essential for analysis in real-time distributed systems [24]. However, there are some techniques [6], [7] that consider the strict timing constraints while analyzing the system for faults but they are more application specific and focus mainly on identifying faults in aactuators in automotive vehicles. Whereas our application focuses on the general domain of active diagnosis and proposes diagnosis in heterogeneous distributed systems. The motivation for our diagnostic approach comes from the work done by authors in [24] where they have used dynamic semantic web services to enable active diagnosis in open distributed embedded systems. A previous diagnostic technique was introduced by us in [25] where diagnostic queries were scheduled to identify faults in homogeneous distributed embedded systems. The present work improves the aforementioned method by considering faults in heterogeneous distributed embedded systems.

The performance and efficiency of any distributed system critically depends upon the method used to schedule the application tasks on the free processors [26]. Similarly, the schedule of a diagnostic application is really important for the timely fault analysis of the system. It is essential to find the points in time at which each diagnostic task is executed to bound the time required for inferring faults. The creation of efficient task schedules becomes more critical when the corresponding distributed system consists of heterogeneous processors as the scheduler has to consider various computation costs of the same task on different processors in addition to the overhead interprocessor communication. A faulty decision in the schedule may limit the performance and efficiency of the heterogeneous distributed system [26].

Task scheduling in distributed systems is generally shown as a NP-complete problem. Therefore many heuristic based scheduling algorithms have been proposed [26]–[35]. One of the most studied class of scheduling heuristics is the list based scheduling [36]. A list-based scheduler works in two main steps. Firstly, according to some specific criterion, priorities are assigned to the application tasks. Then, depending upon the sorting criteria, the ready task with the highest or lowest priority is scheduled on to a free processor such that the allocation minimizes the predefined cost function. This step is then continuously repeated until all the application tasks are scheduled. At the end of this process, a valid schedule is obtained. Some examples of list-based scheduling algorithms are: dynamic critical path (DCP) [29], heterogeneous earliest finish time (HEFT) [32], critical-path on processor (CPOP) [32], critical-path on cluster (CPOC) [33], critical path/most immediate successors first (CP/MISF) [34], modified critical path (MCP) [35], contention aware scheduling methods [28], [30] and longest dynamic critical path (LDCP) [26]. The mentioned techniques are generally used for scheduling aperiodic application tasks on a distributed embedded system. Since our application model considers periodic tasks so the list based scheduling technique has to be modified to fulfill the requirements.

The goal of the present work is to incorporate fault analysis in a heterogeneous distributed system that has limited resources and stringent timing constraints. The system identifies faults using a graph of queries that is modeled on resource description framework data. This model has been previously used by authors in [24], [25]. Each query is associated with a sensor or group of sensors that provide data in a cyclic order. These sensors utilize local-error identification methods, e.g. message classification, to provide diagnostic facts of the system that serve as the starting point for the analysis. For controlled analysis, these facts are divided into intermediate steps called symptoms. Each fact is realized as a query in the real-time database and is used for the generation of symptoms or identification of faults. The output data from the sensors is stored in the real-time database, and is timely and consistently replicated to ensure distributed execution of the queries. As scheduling the diagnostic tasks on to the system is an important aspect of this fault analysis, so the present work proposes a time-triggered scheduler for a heterogeneous distributed system that uses diagnostic queries and a real-time database to find faults. The purpose of the scheduler is to guarantee fault identification within predictable time while respecting the precedence and periodicity constraints of the diagnostic tasks. Prior to the execution of the diagnostic application, the proposed algorithm calculates the points in time at which each query task is executed and data is replicated to or deleted from the data-base to make the system applicable in a real-time domain [24]. The algorithm uses a specific priority scheme to incorporate the heterogeneity of the system and schedules the diagnostic tasks while respecting their precedence and periodicity constraints.

The rest of the paper is structured as follows. Section II defines the scheduling problem and discusses the application and architecture model used for the analysis. Section III illustrates the scheduling algorithm whereas the simulation results are presented in the Section IV. Lastly, Section IV concludes the discussion and also discusses some ideas for future work.

II. Problem Definition

The present article addresses fault analysis in a heterogeneous distributed environment that has limited resources and stringent timing constraints. The distributed system consists of a group of sensors that provide continuous data in a cyclic order. These sensors utilize error identification methods, like message classification, to provide facts that act as the starting point for analysis. The output data from the sensors is stored in a real-time database and each fact is then realized as a query in this real-time database. The queries are modeled on resource description framework (RDF) and are used for the generation of intermediate analysis steps known as symptoms

and for the identification of system faults. This type of model has been previously used by authors in [24].

A. Application Model

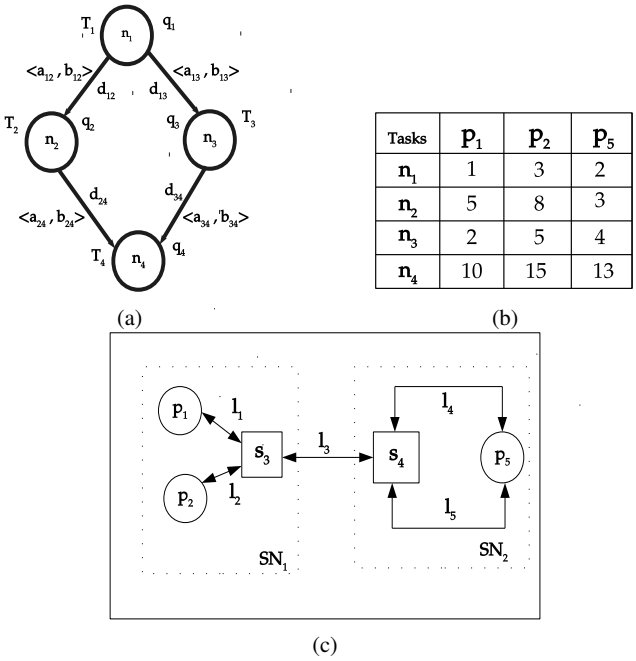
The diagnostic framework used in this scenario requires the static scheduling of queries on a set of m heterogeneous processors. The application model is represented by a diagnostic graph of queries. This graph is formally termed as a *diagnostic multi-query graph* (DMG). In this graph, each query is unique in nature and is linked with a sensor or a specific set of sensors. The sensors provide data in a periodic manner, which is then stored in the real-time database and is timely replicated on the end systems to ensure the distributed execution of the queries. The queries are used for the generation of symptoms or to identify faults and are executed periodically to keep the diagnosis up-to-date with the sensors data.

In a DMG, each node represents a single query and each edge specifies the input/output relationship between two queries through the real-time database i.e. the precedence constraints and the parent-child relationship between the queries. The nodes are termed as features (nodes without incoming edges), symptoms (nodes with both incoming and outgoing edges) and faults (nodes without outgoing edges) respectively. Since the sensors are sending data periodically so it is essential for the queries to be executed after continuous intervals to keep the analysis up-to-date. Therefore, each query is linked with a strict time period. Since the cyclic execution of queries result in various iterations so it is possible that the child node requires output data from one or a group of the previous iterations rather than the present one. This feature is incorporated in the DMG in the form of history interval and has been explained in detail in later sub-sections.

Such a graph can be represented by:

$$G = (N, E, q, T, d) \tag{1}$$

where each node $n_i \in N$ is a non-divisible periodic query task. Each diagnostic edge $e_{ij} \in E$ represents the relationship between node $n_i \in N$ and node $n_j \in N$ such that n_i is the parent node to n_j. Query q_i represents the query associated with the node $n_i \in N$. Period T_i represents the period after which the query q_i has to be executed. This period T_i also represents the deadline constraint for the query q_i. Data d_{ij} on edge e_{ij} $\in E$ represents the amount of output tuples transmitted from node $n_i \in N$ to $n_j \in N$. The edges also represent the history interval $< a_{ij}$, $b_{ij} >$ where a_{ij} represents the start of the data transference from the parent node n_i to the child node n_j and b_{ij} represents the end of the data transference. The subsequent child node n_j cannot start its execution before all the data has been transmitted from the parent node n_i. This has been explained in detail in Section IIC. The size of the DMG is not restricted and it can have any arbitrary size and shape. An example of such a DMG is given in fig. 1a. We assume that the structure and characteristics of the application graph are completely known prior to execution. In the later sections, each node $n_i \in N$ may be referred to as a query task or simply task of the graph G represented by (1).



Tasks	p_1	p_2	p_5
n_1	1	3	2
n_2	5	8	3
n_3	2	5	4
n_4	10	15	13

Fig. 1: a). Diagnostic multi-query graph - G b). Computation matrix W and c). Example of the architecture model

B. Architecture Model

The distributed model used for the analysis is comprised of SN sub-nodes. Each sub-node has P processors, S switches and L bi-directional links between them. The sub-nodes are connected with each other through bi-directional communication links to make a fully connected distributed system. Each sub-node SN_i can be represented by (2).

$$SN_i = (P, S, L) \tag{2}$$

where each $p_i \in P$ is a heterogeneous processor and each $s_i \in S$ represents a switch in the sub-node. The bi-directional link $l_{p_i s_i} \in L$ shows a connection between processor p_i and switch s_i whereas each $l_{s_i s_j} \in L$ represents a link between switches s_i and s_j. It is to be noted that there is no direct connection between two processors in the system. The overall parallel system is represented by a m x n computation cost matrix W. The rows in W represent all the processors of the system whereas the columns represent the query task nodes of the application graph G represented by (1). Each entity $w_{ij} \in W$ represents the estimated computation cost of the node $n_i \in N$ when it is executed on the processor $p_j \in P$. Table in fig. 1b shows the computation cost matrix for the graph in fig. 1a when it is executed on the parallel system shown in fig. 1c.

The overall parallel system has following characteristics:

- The system has heterogeneous processors, routers and links.
- Each processor $p_i \in P$ has its own sensor or set of sensors that are providing input data periodically to the system.

- The system has a distributed database i.e. each processor $p_i \in P$ has its own real-time database that is timely replicated on other processors in P to carry out the distributed executions of the queries.
- The system runs both the target and diagnostic applications in parallel to each other without any conflict between them. The scheduler is central to the system and has its own dedicated resources to compute the schedule which do not take part in the execution of the diagnostic multi-query graph (DMG).
- The processors in the system are designed for the execution of the queries only and do not take part in any kind of communication. Similarly, the switches are available for the transmission of the messages only and they have no part in the execution of the queries.
- The scheduling is non-preemptive i.e. once a query task is assigned to a processor, it gets free only when the task has completed its execution. A processor executes one task at a time only and all the periodic iterations of a task are executed on the same processor. Moreover, the computation cost, once assigned to a task, remains same throughout the schedule.
- Each link in the system has a fixed rate of transmission. The cost of transmitting a message over a link is calculated using (3).

$$c_{l_k}(m_{ij}) = \frac{d_{ij}}{B_{l_k}} \quad \forall (n_i, n_j) \in N, n_i \in pred(n_j), l_k \in L \quad (3)$$

Here $c_{l_k}(m_{ij})$ is the cost of transmitting message of size d_{ij} from task n_i to n_j over a link l_k that has a transmission rate of B_{l_k} units. We are assuming that data is transmitted on all links at the same time. It is a safe assumption as in today's parallel systems there is generally no significant delay between two adjacent links. However, to correctly incorporate the heterogeneity of the links, it is essential that the start time of an edge on a link is not earlier than its start time on the first link of the path and its finish time is not earlier than that of the previous link in the path. For instance, a message m_{12} of 4 units is transmitted from p_1 to p_2 on a path containing links $< l_1, l_2 >$ having bandwidths of 2 and 4 units respectively. Assume that m_{12} is ready for transmission at 1 unit and l_1 is free to transmit it. So the message starts its execution on l_1 at 1 unit and completes it at 3 units (duration = $\frac{4}{2}$ = 2). Now if m_{12} starts its transmission on l_2 also at 1 unit then it will reach the corresponding processor at 2 units (duration = $\frac{4}{4}$ = 1) which results in loss of data as the previous data has not been transmitted yet. To make sure that it does not happen m_{12} should start its transmission on l_2 later than its transmission on l_1. Eq. (4) is used from [30], to calculate the minimum time at which a message m_{ij} should start its transmission on a link l_k in the path $< l_1, l_2, ..., l_k, ..., l_n >$ to avoid any possible loss of data.

$$\begin{aligned} ST_{min}(m_{ij}, l_k) = max\{ST_{act}(m_{ij}, l_1), \\ FT_{act}(m_{ij}, l_{k-1}) - c_{l_k}(m_{ij}) \quad \forall \quad k \neq 1\} \end{aligned} \quad (4)$$

Here, $ST_{act}(m_{ij}, l_1)$ is the actual start time of m_{ij} at l_1, $FT_{act}(m_{ij}, l_{k-1})$ is the time at which m_{ij} finishes its execution on $kth-1$ link of the path and $c_{l_k}(m_{ij})$ is the communication cost of the message m_{ij} on kth link of the path. Lastly, $ST_{min}(m_{ij}, l_k)$ is the minimum possible time at which the message m_{ij} can start its execution on link l_k of the path.

- The path for the message transmission is selected using the shortest path routing policy. The algorithm finds all possible paths among the two processors and then arranges them in increasing order of their costs. Then the algorithm selects the first available path from this list for the transmission of the message. Similar to processors, a path transmits only one message at a time. All the iterations of a message are executed on the same path and their communication cost remains same throughout the schedule.
- The communication is carried out through a deterministic protocol such as Time Division Multiple Access (TDMA).
- If two dependent tasks are assigned to the same processor then their communication cost is negligible. It is in regard to the fact that in parallel systems, the cost of inter-communication is generally much higher than that of intra-communication. If the dependent tasks are assigned to two different processors, then the communication cost is calculated using (3).

There is no restriction on the structure of the parallel system and it can follow any topology with an arbitrary number of sub-nodes and with an arbitrary number of communication links between them. An example has been shown in 1c.

C. Definitions

This section will elaborate some of the basic terms that will be used throughout the article.

- ***Time Period:*** Since the sensors in the system are providing input data after specific intervals so each query task is linked with a strict period to keep the analysis up-to-date. It means that in a task graph G represented by (1), each node $n_i \in N$ will repeat itself after a strict interval T_i. This interval is termed as time period and is the exact time difference between two consecutive iterations of a query task [37]. It can be represented by (5).

$$T_i = s_{i_{k+1}} - s_{i_k} \quad \forall n_i \in N \quad (5)$$

where T_i is the time period of node $n_i \in N$ whereas s_{i_k} and $s_{i_{k+1}}$ are the start times of the kth consecutive iterations of $n_i \in N$. The time period T_i also serves as a form of deadline for the node $n_i \in N$ i.e. each iteration of n_i should complete its execution before the start time of its next iteration that is calculated using T_i. This deadline can be represented as,

$$s_{i_k} + w_{ij} < s_{i_k} + T_i \quad \forall n_i \in N \quad (6)$$

where w_{ij} is the computation cost of node n_i on processor $p_j \in P$. It has to be noted here that the computation cost

w_{ij} of task $n_i \in N$ on processor $p_j \in P$ should always be less than the time period T_i of n_i for the successful execution of the task n_i on p_j within its deadline. The factor $\frac{w_{ij}}{T_i}$ is known as the node utilization factor, γ_{ij}.

$$\gamma_{ij} = \frac{w_{ij}}{T_i} < 1 \quad \forall n_i \in N, T_i \in G, p_j \in P \tag{7}$$

The time periods linked to the nodes in the graph G are also used to calculate the number of times a parent node has to repeat itself before its corresponding child node is ready for its first execution. The formula for this calculation is represented by (8). Here, $n_p \in N$ is the parent node of $n_c \in N$, num_{pc} is the number of times n_p should be repeated for n_c to start its execution and T_c and T_p are the time periods of n_c and n_p respectively.

$$num_{pc} = \begin{cases} \frac{T_c}{T_p}, & \text{if} \quad T_c \bmod T_p = 0 \cap T_c \geq T_p \\ \frac{T_c}{T_p} + 1, & \text{if} \quad T_c \bmod T_p \neq 0 \cap T_c > T_p \\ 1, & \text{if} \quad T_c < T_p \end{cases} \tag{8}$$

- ***Hyper Period:*** For a set N of n periodic tasks in graph G represented by (1), the hyper-period H_G is the minimum time interval after which G will start its next cycle of executions [37]. It is calculated by taking the least common multiple of the time periods of all the nodes in N.

$$H_G = L.C.M.(T_1, T_2, T_3, ..., T_n) \tag{9}$$

The hyper-period is an important characteristic of a periodic task graph G (represented by (1)) and is used to calculate the minimum number of times each task is repeated within one complete execution [27] as shown in (10).

$$num_i = \frac{H_G}{T_i} \quad \forall n_i \in N, T_i \in G \tag{10}$$

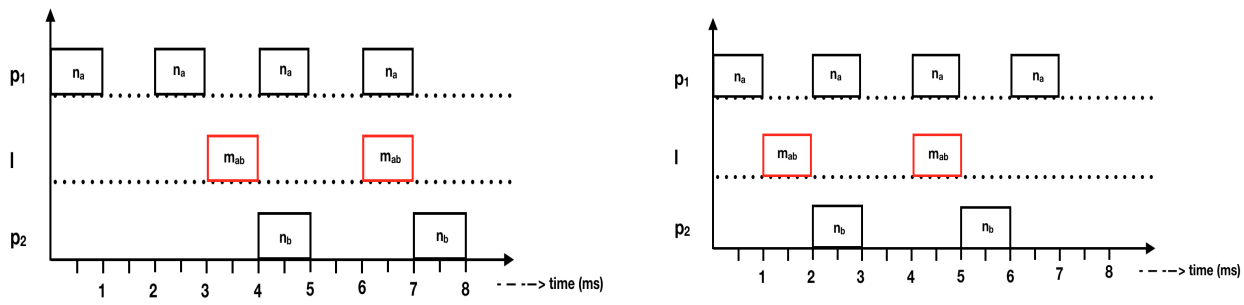
where, H_G is the hyper-period of the graph G, T_i is the time period of the node $n_i \in N$ and num_i is the number of times n_i has to be repeated within one complete execution of the graph.

- ***History Interval:*** As mentioned in Section II(A), each edge $e_{ij} \in E$ is linked with a history interval $< a_{ij}, b_{ij} >$ to incorporate the multi-iteration dependencies between tasks $n_i \in N$ and $n_j \in N$ where $n_i \in pred(n_j)$. A history interval is essential to determine the amount of data a child task requires as well as the point in time at which it is ready for execution. The start of the history interval determines when data can be discarded from the database and the end of the interval determines the minimum start time of the child node. In simple words, the history interval determines the set of iterations of the parent node required by the child node. Consider a two node graph where n_a is the parent of n_b i.e. n_a is transmitting data, d_{ab} = 1 unit, to n_b. Lets assume that the time period of n_a is 2 units and the time period of n_b is 3 units. The graph is being scheduled on a two processor, p_1 and p_2, distributed system. The computation cost of the nodes n_a and n_b is one unit on both p_1 and p_2. There is only one link l (B_l = 1 unit/ms) between p_1 and p_2. Node n_a is scheduled on processor p_1 while node n_b is scheduled on processor p_2. Now according to (9), the hyper-period H_G for this graph is six units, therefore according to (10), n_a is repeated thrice while n_b is executed twice in one complete iteration of the graph. In Fig. 2, two different cases of history intervals are presented. In the first case, the history interval is $< 0,0 >$ so n_b requires the latest iteration of n_a that can be calculated using (8). In the second case, it is $< 1,1 >$ so the iteration prior to the latest iteration is required for n_b to be ready for its execution. The iterations required by the child node can be represented by a set $< s_1, s_2 >$, where s_1 is the first and s_2 is the last iteration required by n_b to start its execution. It can be calculated using the following equation,

$$< s_1, s_2 > = \begin{cases} < num_{pc} - a_{ij}, num_{pc} - b_{ij} >, \\ \quad \text{if} \quad 0 \leq a_{ij} \leq b_{ij} < num_{pc} \\ < 0,0 >, \quad \text{otherwise} \end{cases} \tag{11}$$

Here, $s_1 = s_2$ when $a_{ij} = b_{ij}$. In the example, n_a is the parent node while n_b is the child node with a time period greater than that of the parent node, so according to (8), $num_{pc} = \frac{T_c}{T_p} + 1 = \frac{3}{2} + 1 = 1 + 1 = 2$. So, for $< 0,0 >$ using (11), $s_1 = s_2 = 2$. For $< 1,1 >$ and for $s_1 = s_2 = 2 - 1 = 1$. It means that for $< 0,0 >$, n_b can start its execution after the second iteration of n_a, in this case the first iteration of n_a is not required to transfer any data to n_b. Whereas, for the second interval n_b is ready after the first instance of n_a. In fig., 2, it is interesting to see how the scheduling length varies with the history interval. If the resultant $< s_1, s_2 > = < 0,0 >$, then it means that the child does not depend upon its parent node. For example, if both the parent and child node have same time periods then the only possible correspondence between them is one-on-one therefore in this case any history interval other than $< 0,0 >$ nullifies the dependency of the child node. For this scenario, the edge between the parent and child node can be removed from the graph. To incorporate proper dependency, it is essential that $0 \leq a_{ij} \leq b_{ij} < num_{pc}$. The scheduling algorithm uses (11) to calculate the starting point of the first iteration of a node after which the node repeats itself according to its time period.

- ***Schedule Length:*** The schedule length SL for a task graph G represented by (1), is the finish time of the last iteration l of the exit node n_e in G [36]. If there are k exit nodes $< n_{e_1}, n_{e_2}, ..., n_{e_k} >$ in G then the maximum finish time among the last iterations of all the exit nodes equates to the schedule length of the graph. It is represented by (12). Here $t_{f_l}(n_{e_1})$ is the actual finish time of the last iteration of the first exit node n_{e_1}, $t_{f_l}(n_{e_2})$ is the finish time of the last iteration of the second exit node n_{e_2} and so on. The purpose of the scheduler is to minimize this schedule



(a) (b)

Fig. 2: a). Case 1 : $< a_{n_{ab}}, b_{n_{ab}} > = < 0,0 >$ and b). Case 2 : $< a_{n_{ab}}, b_{n_{ab}} > = < 1,1 >$

length so that a fault is identified within minimum time.

$$SL = max_{n_e \in N} \left\{ t_{f_l}(n_{e_1}), t_{f_l}(n_{e_2}), ..., t_{f_l}(n_{e_k}) \right\} \quad (12)$$

- ***Path Length:*** For a diagnostic multi-query graph G represented by (1), a path p from node $n_0 \in N$ to node $n_i \in N$ is a sequence of nodes $< n_0, n_1, ..., n_i > \in N$ such that they are connected by edges $< e_{01}, e_{12}, ..., e_{(i-1)i} > \in E$. A path length p is the sum of the weights of the nodes and edges of the path $p \in G$ and can be represented by (13) [36]. Here w_n is the computation cost of the node n and d_e is the amount of data being transmitted on the edge e.

$$pl_p = \sum_{n \in p,N} w_n + \sum_{e \in p,E} d_e \quad \forall p \in G \quad (13)$$

A path that has the greatest path length in G is termed as its critical path cp. A critical path $cp \in G$ always starts at the entry node $n_{entry} \in N$ and ends at the exit node $n_{exit} \in N$. The length of this path usually serves as the lower bound of the schedule length of the graph G.

$$pl_{cp} = max_{p \in G} \left\{ pl_p \right\} \quad (14)$$

- ***Top Level:*** For a diagnostic multi-query graph G represented by (1), top level tl_n of an arbitrary node $n \in N$ is the longest path from the entry node $n_{entry} \in N$ to n [36], top level tl_n is the longest path to n from one of these entry nodes. It is calculated excluding the weight w_n of n. Top level is generally used to attribute priorities to the nodes in a list scheduler. It is mostly used for the graphs that are not completely known at the start of schedule computation. Our scheduler uses this priority attribute to accommodate the heterogeneity of the physical system. It can be calculated using (15).

$$tl_n = max_{n_i \in pred(n) \cap entry(G)} \left\{ pl_{n_i -> n} - w_n \right\} \quad (15)$$

Here w_n is the computation cost of $n \in N$, n_i belongs to the parent nodes of n and $pl_{n_i -> n}$ is the path length from the entry node n_i to n. It has to be noted that if $pred(n) = \emptyset$ then $tl_n = 0$.

- ***Bottom Level:*** For a diagnostic multi-query graph G represented by (1), bottom level bl_n of a node $n \in N$ is the longest path in the graph from n to $n_{exit} \in N$ [36]. It can be calculated using (16). This attribute is used by the scheduler to break ties.

$$bl_n = max_{n_i \in (desc(n) \cap exit(G))} \{ pl_{n -> n_i} \} \quad (16)$$

where *desc(n)* is a set of descendants of task n and $pl_{n->n_i}$ is the path length from the parent node n to the exit node n_i. If *desc(n)* = $\emptyset$ then $bl_n = w_n$.

III. Scheduler

The basic structure of our algorithm is similar to any other list scheduler i.e. assign priorities to the tasks and then schedule them on the free processors. Our scheduler works in four main steps: i). Task selection, ii). priority assignment and ordering the ready list, iii). path and processor selection and lastly iv). status update. A psuedo-code representation of the algorithm is given in fig., 5.

There are some initial steps that need to be performed before the task selection phase.

- Firstly, the algorithm makes sure that the graph can be scheduled. For this purpose, the node utilization factor (7) is calculated. If the utilization factor γ_{ij} of a node $n_i \in N$ on a processor $p_j \in P$ is greater than one then the graph cannot be scheduled as this node will never fulfill its deadline.
- If the graph is schedule-able then the algorithm calculates the hyper-period H_G using (9).
- The algorithm then calculates the number of times each node n_i has to be repeated by using (10). H_G is calculated in step 2.
- A parent-child relationship matrix Ω is computed using (8) and (11). Each entity $\omega_{ji} \in \Omega$ tells the number of the iterations of the parent node n_i required by the child node n_j. This entity represents the relationship within one hyper-period of the parent and child node. The rows in Ω represent the child nodes while the columns represent the parent nodes. When ω_{ji} = 0, it means that there is no relationship between the parent and child node. An example is given in fig., 6d.

- The average computation cost $\overline{w_i}$ of each node $n_i \in N$ is computed using (17). Here, P is the total number of processors in the system.

$$\overline{w_i} = \frac{\sum_{k=1}^{P} w_{i,k}}{P} \tag{17}$$

- Average bottom level $\overline{bl_i}$ of each node $n_i \in N$ is computed using (16) where $\overline{w_i}$, calculated in the previous step, is used for each node.

A. Proposed Algorithm

1) ***Task Selection***: After completing the initial steps of the algorithm, the scheduler computes the ready list R. A node $n_i \in N$ is said to be ready when all its parent nodes have completed their execution [36]. The matrix Ω computed in step 4 of the algorithm is used to determine the iterations of the parent nodes required by the child nodes. If each $n_i \in parent$ of n_j has completed the number of iterations mentioned in Ω, then the node n_j is ready for execution.
2) ***Priority assignment and ordering the ready list***: The top-level of each node $r_i \in R$ is calculated using (15). The ready list R is then ordered in increasing order of the top-level of each node. If priorities are same for two nodes, then the tie is broken using the average bottom level of the nodes computed in step 6 of the algorithm. In this case the node that has greater bottom-level is placed higher in the ready list.
3) ***Path and processor selection***: After the generation of ready list, the scheduler traverses through the processors and paths to select the pair that gives the least computation cost to a task. The goal of this phase is to select the combination that gives the earliest finish time to the scheduling task. Here, we assume that intra-communication cost is zero i.e. if two dependent tasks are scheduled on the same processor then their data transference cost is zero. On the other hand if two dependent tasks are scheduled on two different processors then a message task is created whose communication cost depends upon the path selected for communication. Since both of the tasks are periodic in nature so the data is transferred between them after regular intervals. This interval i.e. time period of the message task is equivalent to the time period of the sending task if it is greater than that of the receiving task and vice versa.
    - ***Path selection***: If a message m_{ij} is being transmitted from processor p_i to p_j, then among all the paths between p_i and p_j, the path $path_{ij}$ that gives the least communication cost is selected. The finish time of a message m_{ij} on path p_{ij} is the time at which m_{ij} completes its transmission on the last link l_k of the path (18). Here, $FT(m_{ij}, l_k) = ST(m_{ij}, l_k) + c_{l_k}(m_{ij})$. $ST(m_{ij}, l_k)$ is the start time of m_{ij} on l_k (4) and $c_{l_k}(m_{ij})$ is the duration of m_{ij} on link l_k (3). The path that gives $\min_x \{FT(m_{ij}, path_x)\}$ is selected where x represents the total number of paths between p_i and p_j.

$$FT(m_{ij}, path_{ij}) = FT(m_{ij}, l_k) \tag{18}$$

    - ***Processor selection***: The scheduler traverses through all the processors of the system and selects the processor that gives the earliest finish time to the scheduling task. The finish time of n_i on a processor p_j is given as [30]:

$$FT(n_i, p_j) = ST(n_i, p_j) + w_{ij} \tag{19}$$

    Here, $ST(n_i, p_j)$ is the start time of n_i on p_j and w_{ij} is the computation cost of n_i on p_j. So the processor that gives the minimum finish time is selected.

    While scheduling a task to a processor or a message to a link, it has to be made certain that the task or message are not hindering the execution of the other tasks or messages already assigned on the processor or link. This process has been explained in fig., 3 and 4. Note that all the iterations of the scheduled task are executed on the same processor and the task repeats itself according to its time period. It is similar in the case of message assignment to paths.
4) ***Status Update***: Once the task is scheduled on a processor, the status of the system is updated. As n_i is now scheduled on p_j so its computation cost is no longer unknown. Hence the weight of the node that identifies n_i in the graph is set to its computation cost on p_j. Moreover, the weight of the edges between n_i and all of its parents that were scheduled on p_j are set to zero. This is really important because the assignment of a parent and child node on the same processor might change the critical path of the graph in the next step as this edge is still included in the calculation of the top-level of the next node. So, in order to compute the correct top-level, the weight of the edge is updated to zero.

The above mentioned steps are repeated until a valid schedule is obtained. As an example consider the DMG and architecture

```
1: w_{j,i} ← Computation cost of node n_j on processor p_i
2: T_j ← time period of node n_j
3: k ← Iteration number of node n_j
4: l ← Iteration number of node n_i already scheduled on processor p_i
5: H_G ← Hyper-period of the graph G
6: S_j[k] ← Start time of the kth iteration of node n_j on processor p_i
7: S_i[l] ← Start time of the lth iteration of node n_i on processor p_i
8: w_{i,i} ← Computation cost of node n_i on processor p_i
9: T_i ← time period of node n_i
10: for k = 0 to k = H_G/T_j do
11:     duration = S_j[k] + w_{j,i}
12:     for l = 0 to l = H_G/T_i do
13:         d = S_i[l] + w_{i,i}
14:         if (S_i[l] ≥ S_j[k] < d) OR (S_i[l] ≥ duration < d) then
15:             n_j can not be scheduled on p_i
16:             Exit
17:         end if
18:     end for
19: end for
```

Fig. 3: Algorithm to check if a node n_j can be scheduled on the processor p_i

```
1: w_j ← Communication cost of message m_j on link l_i
2: T_j ← time period of message m_j
3: k ← Iteration number of message m_j
4: n ← Iteration number of message m_i already scheduled on link l_i
5: H_G ← Hyper-period of the graph G
6: S_j[k] ← Start time of the kth iteration of message m_j on link l_i
7: S_i[n] ← Start time of the nth iteration of message m_i on link l_i
8: w_i ← Communication cost of message m_i on link l_i
9: T_i ← time period of message m_i
10: for k = 0 to k = H_G / T_j do
11:     duration = S_j[k] + w_j
12:     for n = 0 to n = H_G / T_i do
13:         d = S_i[n] + w_i
14:         if (S_i[n] ≥ S_j[k] < d) OR (S_i[n] ≥ duration < d) then
15:             m_j can not be scheduled on l_i
16:             Exit
17:         end if
18:     end for
19: end for
```

Fig. 4: Algorithm to check if a message m_j can be scheduled on the link l_i

```
1: for each node n_i ∈ N do
2:     for each processor p_j ∈ P do
3:         if w_ij / T_i ≥ 1 then
4:             Exit the schedule.
5:         end if
6:     end for
7: end for
8: Calculate the hyper-period H_G of the graph.
9: Calculate the number of times each node n_i is repeated within one cycle of
   the graph, num_i.
10: Compute the parent-child relationship matrix Ω.
11: Compute the w_i of each node n_i ∈ N.
12: Compute bl_i of each node n_i ∈ N.
13: while there are unscheduled tasks do
14:     Compute the ready-list R.
15:     for each ready node r_j ∈ R do
16:         Compute tl_j.
17:         Order the tasks in increasing order of their top-level. The ties are
            broken by considering the average bottom-level of the node.
18:         for each processor p_j ∈ P do
19:             for each parent node n_i ∈ parent of r_i do
20:                 if p_{n_i} ≠ p_j then
21:                     Create a communicating message m_ij.
22:                     for each path path_k between p_{n_i} and p_j do
23:                         for each link l_m in path_k do
24:                             if m_ij can be scheduled on l_m then
25:                                 Calculate the finish time of m_ij on l_m
26:                             end if
27:                         end for
28:                         The communication cost of m_ij on path_k is the maxi-
                            mum time the last link in the path has taken to transmit
                            the information.
29:                     end for
30:                     Select the path that gives the earliest finish time to m_ij.
31:                 end if
32:                 if r_j can be scheduled on p_j then
33:                     Calculate the finish time of r_j on p_j
34:                 end if
35:             end for
36:         end for
37:         Select the processor that gives the earliest finish time to r_j.
38:     end for
39:     Update the weights of the edges and the already scheduled nodes on the
        graph.
40: end while
```

Fig. 5: Pseudo-code representation of the list scheduling algorithm

model given in fig., 1a and 1c respectively. The schedule generated by the algorithm is given in fig., 7. Since n_4 completes its second iteration on 16 ms so SL = 16ms.

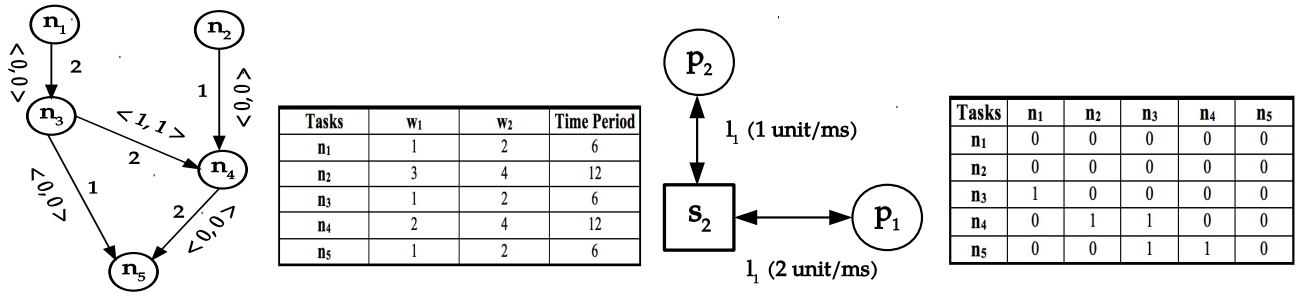
IV. Experimentation and Analysis

This section illustrates the setup used for implementing the scheduling algorithm, implements the algorithm in different scenarios and compares the corresponding results. In the mentioned list scheduling algorithm, the time-complexity for finding bottom and top levels of nodes using depth first search (DFS) is $O(N+E)$ [38], sorting the nodes according to their priorities has a complexity of $O(NlogN)$, assigning the communication task to the best possible link i.e. the path that gives the earliest finish time to the message has a complexity of $O((N+E)PL)$ and lastly assigning the node to the processor that gives its earliest finish time has a complexity of $O(N+E)P$. It shows that the scheduler highly depends upon the size of the application graph as well as the structure of the architecture model. The highest complexity is for the last two steps of the scheduler. It can be reduced by restricting the algorithm to consider the first free link or the first available processor instead of searching for the best possible solution. This, however, might increase the schedule length of the graph. This is a scenario that will be discussed in future works.

A. Experimental Setup

For experimental purposes, the algorithm is programmed in C language and is implemented on a linux based on-line server. A set of randomly generated graphs are used to test the scheduler and the results are compared through varying parameters. The following characteristics are used to generate these test graphs.

1) The total number of nodes in the DMG represented by N.
2) Number of edges per node in the DMG represented by $Edges_n$.
3) Time period T_n of a node n in the graph. It is randomly generated from the arithmetic sequence $< T_{min}, T_{min}+1, ..., T_{max} >$ such that the average time period $\overline{T}$ of the graph is always equal to $\frac{T_{min}+T_{max}}{2}$.
4) History interval $< a_{ij}, b_{ij} >$ between a node n_i and n_j, where n_i is the parent of n_j, in the graph is generated randomly such that $0 \leq a_{ij} \leq b_{ij}$ ¡ num_{ij}. Where num_{ij} is calculated using (8) using time-period T_i of n_i and time-period T_j of n_j. The history interval is always taken as a whole number rather than a fraction.
5) Average node utilization factor, $\overline{\gamma}$: The ratio of average computation cost of a node to its time period. This factor is used to calculate the average computation cost $\overline{w_n}$ of a node. The lower the value of $\overline{\gamma}$, the better will be chances of the node to uphold its deadline. To ensure this scenario, (7) should always hold true.
6) Computation cost heterogeneity factor α: This factor is used in most scheduling algorithms [26], [39] to incorporate the heterogeneity of the system. If $\alpha = 0$ then the computation cost of a node is same for all the processors. If there are p processors in a distributed system and the average computation cost (calculated through $\overline{\gamma}$) of a node n in the graph is $\overline{w_n}$ then the



Tasks	w_1	w_2	Time Period
n_1	1	2	6
n_2	3	4	12
n_3	1	2	6
n_4	2	4	12
n_5	1	2	6

Tasks	n_1	n_2	n_3	n_4	n_5
n_1	0	0	0	0	0
n_2	0	0	0	0	0
n_3	1	0	0	0	0
n_4	0	1	1	0	0
n_5	0	0	1	1	0

(a) (b) (c) (d)

Fig. 6: a). A sample DMG with H_G = 12 b). Characteristics of graph given in (a) when implemented on system given in (c) c). A sample parallel system and d). The parent-child relationship matrix Ω of the graph in (a)

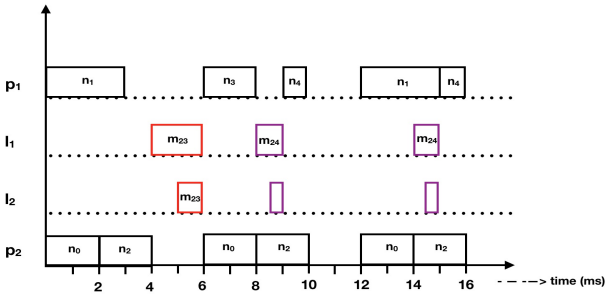


Fig. 7: Time-line representation of the schedule produced by the algorithm when fig. 6a is implemented on fig. 6c

computation cost of n on a processor p_i should lie between $[\overline{w_n}*(1-\frac{\alpha}{2}), \overline{w_n}*(1+\frac{\alpha}{2})]$ [26], [39]. This range is used to calculate p different computation costs of a node n and then the costs are arranged in ascending order i.e. the lowest cost is assigned to the p_1 and so on. The value of α is between $(0, \frac{2-2\overline{\gamma}}{\overline{\gamma}})$ so that the computation cost of a node at any processor is not greater than its time period.

7) Communication to computation cost ratio, *CCR*: The average communication cost divided by the average computation cost of the graph. This factor is used to calculate the amount of data transferred from a parent node to its child node. The average computation cost $\overline{w_n}$ of a parent node n in the graph is multiplied by this ratio to determine the amount of data being transferred to its child nodes. A higher value of this ratio will result in an increase in the amount of data being transmitted.

Different architecture models are generated for the implementation of the scheduler. A single cluster c_i in the model consists of only homogeneous processors therefore heterogeneity factor α is taken as zero for each cluster. These clusters are combined together to form a heterogeneous distributed system. Following characteristics are used to generate these models.

1) Total number of processors used in the distributed system, P.
2) Total number of homogeneous clusters, C, in the distributed system.
3) Network topology used in each cluster and the overall system.
4) Total number of processors num_p in a single cluster, c_i and total number of switches num_s in a single cluster c_i. These characteristics usually depend upon the network topology used within the cluster.
5) Total number of links per node in the homogeneous cluster c_i represented L_t.
6) Rate of transmission B_{l_k} of a link l_k in a cluster c_i. It is randomly generated from the arithmetic sequence $< B_{min}, B_{min}+\beta, ..., B_{max} >$ such that the overall bandwidth of the cluster is always equal to $\frac{B_{max}+B_{min}}{2}$. β is the common difference between two consecutive numbers of the sequence.

B. Results

The randomly generated graph set consists of 300 application DMGs with node size ranging from 200 to 600 with an increment of 200 nodes per graph. For the total number of edges per node, we utilized the value of 4. The time period is taken from the arithmetic sequence $< 1,2,3,...,10 >$ and is measured in milliseconds. The average node utilization factor, $\overline{\gamma}$ is varied between 0.25 and 0.5. These values of $\overline{\gamma}$ result in an upper-bound of 6 and 2 respectively for the heterogeneity factor α so we choose it as 1 which means that the computation cost w_n of a node n on a processor p_i is between $[0.5\overline{w_n}, 1.5\overline{w_n}]$. The average computation cost $\overline{w_n}$ measured using $\overline{\gamma}$ and the time period T_n of the node n is also given in milliseconds. Lastly the communication to computation cost ratio , *CCR* is varied between 0.1, 0.5 and 1.0. For the architecture model, the total number of processors are varied between 16, 32 and 64. There are four processors in each cluster so the total number of clusters vary between 4, 16 and 32. Each cluster has a star topology with a single main switch and there is only one link between each processor and the main link. The bandwidths of the links are generated from the arithmetic sequence $< 4,6,8,10 >$. The rates are

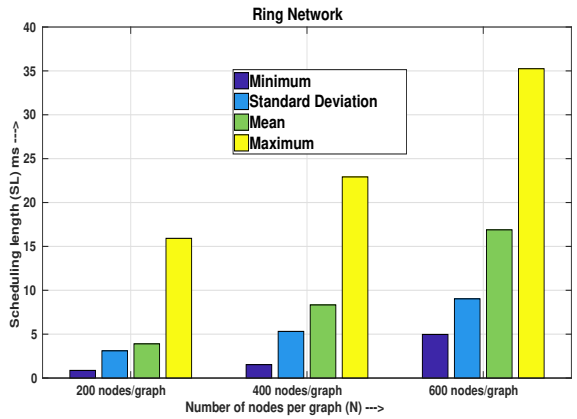
given in units / milliseconds. For the overall system, two topologies are used i). ring topology and ii). fully-connected network. The homogeneous clusters are combined together using these topologies to form a heterogeneous distributed system. The metrics used to explain the results are scheduling length (12) and scheduling rate of the graphs (20). These metrics were measured against the total number of nodes per graph, communication to computation cost ratio (CCR), total number of processors in the distributed system, the network topology of the overall distributed system, and the average node utilization factor.

$$Rate_s = \frac{\text{total number of graphs scheduled}}{N} * 100 \qquad (20)$$

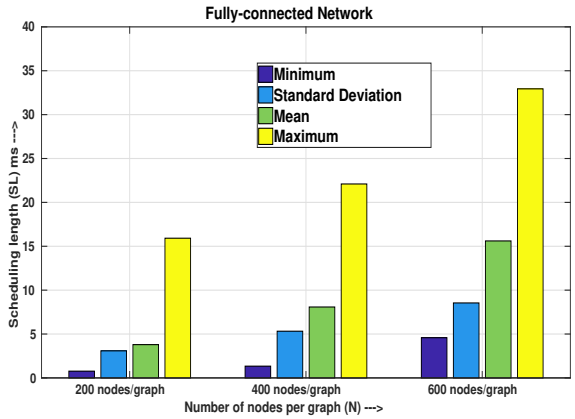
- ***Total number of nodes / graph***, ***N***: The scheduling length increases with an increase in the total number of nodes of the DMG. Consider the bar graph shown in fig. 8a. Here, 100 graphs each of 200 nodes/graph, 400 nodes/graph and 600 nodes/graph are scheduled on a 64-ring networked distributed system. *CCR* and $\overline{\gamma}$ are taken as 0.5 and 0.25 respectively. Each bar in the graph represents the minimum, standard deviation, mean and the maximum values of the scheduling lengths obtained for each multi-query graph set respectively. Here, the average scheduling length for 100 nodes/graph is approximately 8 ms while it increases to approximately 40 ms when the total number of nodes are increased to 600. The graph size does not effect the scheduling rate that much if there are enough resources in the distributed system.
- ***Communication to computation cost ratio***, **CCR**: An increase in *CCR* increases the amount of data being transmitted from a parent to its child node. Which in turn increases the congestion over the network as each communicating task now takes more time to pass over a link. For example, an edge transmitting data of 4 units over a link of rate 4 units/ms takes 1 ms to complete its transmission. Now if the data is doubled to 8 units, the same edge occupies the link for double amount of time. Consider the graph in fig. 9a. Here 30 separate instances of $N = 200$ nodes per graph have been scheduled on a fully connected distributed system consisting of 32 processors. The *CCR* was varied between 0.1, 0.5 and 1.0. Heterogeneity and average node utilization factor are 1 and 0.25 respectively. As is visible from the spikes in the graph, an increase in the *CCR* value increases the overall schedule length. It was also noticed from the results that increasing this *CCR* value often results in a decrease in the scheduling rate over the same network. It is explainable in the sense that a task which was previously scheduled over a link might now be hindering other tasks on that link with its increased duration.
- ***Total number of processors in the distributed system***, ***P***: The increase in number of processors result in higher parallelization of tasks which in turn reduces the over all schedule length of the graph. Consider the fig. 9b. Here a DMG consisting of 400 nodes is scheduled on 16-, 32- and 64-fully connected distributed systems. When a DMG is scheduled on a 64 processor distributed system, the scheduling length is reduced in comparison to when it is scheduled on a 16 processor distributed system. An interesting point here is that after some time the scheduling length starts converging to the same result irrespective of the amount of resources being used. The parallelization of the graph plays a role in this scenario. For instance, consider a graph that has three tasks that can execute in parallel with each other. Now if the number of processors are two, then two of the tasks can run in parallel while the last task has to wait for one of the other two to stop execution. But if the number of resources are increased to three then the tasks can run in parallel to each other. Now even if the resources are increased to five, the graph still utilizes only three processors so it will give the same schedule length as in the previous case. Same principle applies in case of scheduling rate of the graph. If there are more resources then there is more chance of obtaining a scheduling result.
- ***Network Topology***: Fig., **??** shows the fully-connected network based implementation of the experiment performed in fig., 8a. The results show that there is not much difference in the scheduling length of the graphs. However, the scheduling rate changes when the topologies are switched. For example, for 600 nodes/graph the scheduling rate for 64-fully connected network is approximately 86% but it reduces to 75% when the same scenario is applied on 64-ring based distributed network. It might be because there are more link resources in a fully connected network which result in increased chances of successful communication task allocation. This results in better exploitation of the parallel nodes of the graph. Therefore, an increase in the number of resources increase the scheduling rate of the graph.
- ***Average node utilization factor***, $\overline{\gamma}$: The results show that an increase in $\overline{\gamma}$ from 0.25 to 0.5, decreases the rate of scheduling a graph of 600 nodes on a 64-fully connected distributed system, from 86% to 82%. A higher value of $\overline{\gamma}$ results in an increased computation cost which in turn forces the scheduler to allow more processor time to a node. This decreases the chances of scheduling any other node to the processor. The decrease in the number of free processors decreases the scheduling rate of the graph.

V. Conclusion and Future Work

Active diagnosis is a diagnostic technique that monitors the operation of the safety-critical systems at runtime for fault isolation and error recovery. Since such systems are highly safety-critical i.e. their failure can result in loss of life etc., therefore it is essential that the analysis of the diagnostic information and subsequent recovery from the fault is performed within predictable time. As scheduling the diagnostic application is an important aspect of this timely analysis, therefore the present work proposes a time-triggered static scheduler for a

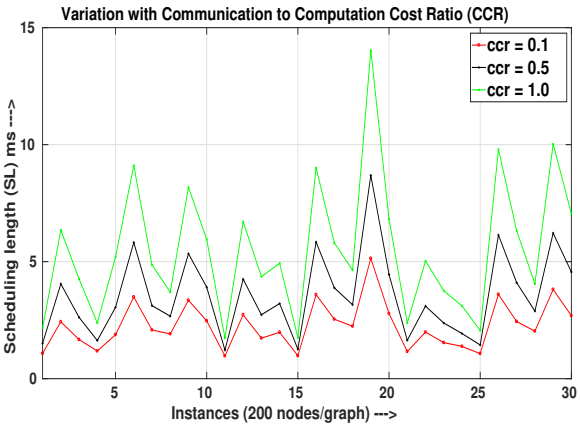


(a) 64-ring, CCR = 0.5, $\overline{\gamma}$ = 0.25, α = 1

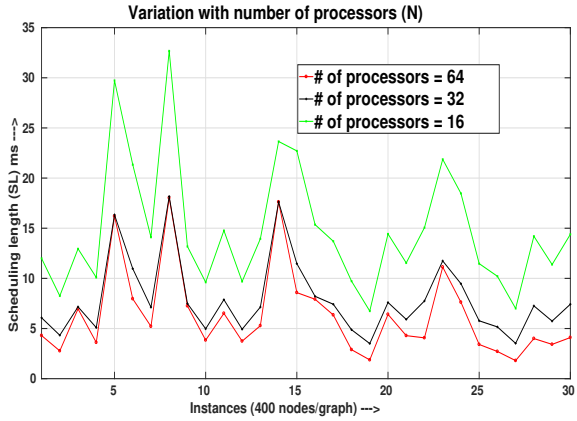


(b) 64-fully connected, CCR = 0.5, $\overline{\gamma}$ = 0.25, α = 1

Fig. 8: Minimum, standard deviation, mean and maximum values of the scheduling lengths of the DMGs



(a) 32-fully connected, 200 nodes/graph, $\overline{\gamma}$ = 0.25, α = 1



(b) 400 nodes/graph, CCR = 0.5, $\overline{\gamma}$ = 0.25, α = 1

Fig. 9: Variation of scheduling length w.r.t. *CCR* and *N*

heterogeneous distributed system that uses diagnostic queries and a real-time database to find faults. The purpose of the scheduler is to minimize the execution time of the query based diagnostic application so that a fault is identified before any major mishap. The proposed algorithm uses the characteristics of a list scheduler where firstly it assigns priorities to each node by calculating their top level, then arranges the ready nodes in ascending order of their priorities to systematically allocate them to free processors that give them the earliest finish time (EFT). Top level is used to incorporate the different computation costs of the nodes throughout the system. In case of equal priorities, the ties are broken using bottom level that is calculated using the average computation cost of each node. If two dependent nodes are scheduled on the same processor then the communication cost between them is negligible but if they are scheduled on two different processors then a communication message is generated whose weight is equivalent to the amount of data being transferred between the nodes. This communication message is then scheduled on the path that gives it the smallest communication cost. The algorithm is prototypically implemented and the results are thoroughly studied and compared on the basis of different characteristics of the system. The results demonstrated in this paper show that a list based time-triggered scheduler can minimize the schedule length of a diagnostic application to increase the performance, reliability and safety of a heterogeneous distributed system.

For future work, our aim is to introduce active diagnosis in open distributed real-time embedded (ODRE) systems. In ODRE systems, design components are dynamic i.e. they leave or enter the system at run-time. It means that the diagnostic graph changes with every change in the distributed system which results in the re-computation of the schedule. The main challenge in this scenario is to keep the schedule as similar as

possible to the previous one. The work presented in this article will be extended to minimize the schedule re-computation in ODRE systems.

ACKNOWLEDGMENT

This work has been supported in part by the European project FP7 DREAMS under project No. 610640 and by the German Research Foundation (DFG) under project ADISTES (OB384/6-1, 629300).

REFERENCES

[1] P. E. Lanigan, S. Kavulya, P. Narasimhan, T. E. Fuhrman, and M. A. Salman, "Diagnosis in automotive systems: A survey," 2011.

[2] E. O. Schweitzer, D. Whitehead, A. Guzman, Y. Gong, and M. Donolo, "Advanced real-time synchrophasor applications," in *proceedings of the 35th Annual Western Protective Relay Conference, Spokane, WA*, 2008.

[3] C. McCann and R. Pigeau, "Clarifying the concepts of control and of command," in *Proceedings of the 1999 Command and Control Research and Technology Symposium*, vol. 29, 1999.

[4] J. C. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 547–550.

[5] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Systems*, vol. 22, no. 5, pp. 64–81, 2002.

[6] P. M. Khilar and S. Mahapatra, "A distributed diagnosis approach to fault tolerant multi-rate real-time embedded systems," in *Information Technology,(ICIT 2007). 10th International Conference on*. IEEE, 2007, pp. 167–172.

[7] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Time-constrained failure diagnosis in distributed embedded systems: application to actuator diagnosis," *IEEE Transactions on parallel and distributed systems*, vol. 16, no. 3, pp. 258–270, 2005.

[8] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Transactions on Electronic Computers*, no. 6, pp. 848–854, 1967.

[9] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *aCM Computing Surveys (CSur)*, vol. 25, no. 2, pp. 171–220, 1993.

[10] R. C. Dorf and R. H. Bishop, "Modern control systems," 1998.

[11] A. Pelc, "Optimal fault diagnosis in comparison models," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 779–786, 1992.

[12] A. Sengupta and A. T. Dahbura, "On self-diagnosable multiprocessor systems: diagnosis by the comparison approach," *IEEE Transactions on Computers*, vol. 41, no. 11, pp. 1386–1396, 1992.

[13] C. J. Walter, P. Lincoln, and N. Suri, "Formally verified on-line diagnosis," *IEEE Transactions on Software Engineering*, vol. 23, no. 11, pp. 684–721, 1997.

[14] A. T. Dahbura, K. K. Sabnani, and W. J. Hery, "Spare capacity as a means of fault detection and diagnosis in multiprocessor systems," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 881–891, 1989.

[15] S. Tridandapani, A. K. Somani, and U. R. Sandadi, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 865–877, 1995.

[16] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*. IEEE, 1991, pp. 74–83.

[17] M. Diaz, G. Juanole, and J.-P. Courtiat, "Observer-a concept for formal on-line validation of distributed systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 900–913, 1994.

[18] F. Jahanian, R. Rajkumar, and S. C. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.

[19] B. Plale and K. Schwan, "Run-time detection in parallel and distributed systems: Application to safety-critical systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 163–170.

[20] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *Computer*, vol. 26, no. 3, pp. 32–41, 1993.

[21] G. Tortora, "Fault-tolerant control and intelligent instrumentation," *Computing & Control Engineering Journal*, vol. 13, no. 5, pp. 259–262, 2002.

[22] J. C. Yang and D. W. Clarke, "The self-validating actuator," *Control Engineering Practice*, vol. 7, no. 2, pp. 249–260, 1999.

[23] G. Heiner and T. Thurner, "Time-triggered architecture for safety-related distributed real-time systems in transportation systems," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. IEEE, 1998, pp. 402–407.

[24] R. Obermaisser, R. I. Sadat, and F. Weber, "Active diagnosis in distributed embedded systems based on the time-triggered execution of semantic web queries," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*. IEEE, 2014, pp. 222–229.

[25] S. Amin and R. Obermaisser, "Time-triggered scheduling of query executions for active diagnosis in distributed real-time systems," in *Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on*. IEEE, 2017, pp. 1–9.

[26] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *Journal of Parallel and distributed computing*, vol. 68, no. 4, pp. 399–409, 2008.

[27] O. Kermia and Y. Sorel, "A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor," in *Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS'07*, 2007.

[28] Y. K. Kwok and I. Ahmad, "Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors," *Cluster Computing*, vol. 3, no. 2, pp. 113–124, 2000.

[29] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on parallel and distributed systems*, vol. 7, no. 5, pp. 506–521, 1996.

[30] O. Sinnen and L. Sousa, "List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures," *Parallel Computing*, vol. 30, no. 1, pp. 81–101, 2004.

[31] M. A. Khan, "Scheduling for heterogeneous systems using constrained critical paths," *Parallel Computing*, vol. 38, no. 4, pp. 175–193, 2012.

[32] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[33] J. Kim, J. Rho, J.-O. Lee, and M.-C. Ko, "Cpoc: effective static task scheduling for grid computing," in *International Conference on High Performance Computing and Communications*. Springer, 2005, pp. 477–486.

[34] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 1023–1029, 1984.

[35] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[36] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.

[37] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[38] A. Rădulescu and A. J. Van Gemund, "On the complexity of list scheduling algorithms for distributed-memory systems," in *Proceedings of the 13th international conference on Supercomputing*. ACM, 1999, pp. 68–75.

[39] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.