Journal of Information Hiding and Multimedia Signal Processing ©2012 ISSN 2073-4212
Ubiquitous International Volume **3**, Number **3**, July **2012**

Scheduling Optimization for Vector Graphics Acceleration on Multiprocessor Systems

Chung-Ping Young

Department of Computer Science and Information Engineering
National Cheng Kung University
1, University Road, Tainan City, Taiwan 701
cpyoung@mail.ncku.edu.tw

Bao Rong Chang

Department of Computer Science and Information Engineering
National University of Kaohsiung
700, Kaohsiung University Rd., Nanzih District, Kaohsiung, Taiwan 811
brchang@nuk.edu.tw

Zhi-Liang Qiu

Department of Computer Science and Information Engineering
National Cheng Kung University
1, University Road, Tainan City, Taiwan 701
eric.qiu09@gmail.com

Received October, 2011; revised July 2012

Abstract. *In recent years, the number of processor cores on one platform has largely increased, while evenly distributing jobs to every processor becomes an important issue. Most previously discussed scheduling situations were well defined in general cases. In this paper, we propose an algorithm, which is modified from Heterogeneous Earliest Finished Time (HEFT), to increase the performance of a homogeneous system. Our proposed algorithm inherits all the advantages of HEFT such as easy implementation, low complexity, high performance, and so on. In general condition, it generates less overhead for scheduling but the output performance still approximate to or even better than the recent modified version of HEFT up to 7%. The multiprocessor scheduling problems are focused on two dimensional vector graphics, and we will discuss how to estimate the processing time, determine the dependency of each sub-graph, map it onto a directed acyclic graph, and then use our proposed algorithm for vector graphics processing.*
Keywords: Multiprocessor Scheduling, Parallel Processing, Load Balancing, Heterogeneous Earliest Finished Time (HEFT)

1. **Introduction.** In the new computer science algorithms, task scheduling or parallel processing is one of the hot topics for discussion, the main spirit of these studies focus on how to shorten the execution time of the tasks as much as possible. The papers which discuss about the CPU scheduling usually ignore the problems that how to identify the dependency of each task and the execution time that they cost. However, in the real-time systems, inaccuracy assumption might lead to a bad or even a wrong result. If the dependency is derived from the arguments passing from parent tasks to children tasks in Directed Acyclic Graphs (DAG), it might be easier to know that all parent tasks are finished or not during the running time, but the reason that makes two nodes have

precedence constraint is not always so simple. In some applications such as 2D vector graphical accelerated programs, the precedent relation of two tasks (form by two images) is resulted from the overlapping of physical position, although they do not have any argument passing through. In this situation, if we use some scheduling algorithms in order to improve the processing performance, we have to check the overlapping between each drawing elements of the entire graph, and then map each task to a DAG, and finally apply the scheduling algorithm to determine the execution order and assign the tasks to the processors. Whenever a task t_i has finished, the responsible processor has to broadcast the result or at least one signal to inform other processors that t_i has been finished so that the immediate children of t_i can be started.

For example, in a task set $T = \{t_1, t_2, t_3\}$ and a processor set $P = \{p_1, p_2\}$, t_2 and t_3 are two images upper crossing over t_1 on some pixels, in other words, for this situation, t_2 and t_3 cannot start until t_1 has been finished. In each multiprocessor system, according to the computational capability of each processor, we classify them into two kinds, homogeneous and heterogeneous systems, both of them are known as a combination of more than one computation unit machines. For exhaustively use the computation units at all the time, there is a huge number of scheduling algorithms such as HEFT [1, 2] has been studied in the world by now, although most of them had discussed about the distributed heterogeneous systems scheduling problems. The simple reason is because we can easily cut down some constraints from an existed heterogeneous scheduling algorithm to form a new homogeneous one, for example, if we assign to each processor the same value for computational capability or even assign each edge weight of DAG by zero, it implies that we intend to use a heterogeneous algorithm for solving the homogeneous systems scheduling problems. Those are the general cases, whereas some heterogeneous algorithms cannot be simply migrated. There is only one thing we want to denote here is that the algorithm HEFT we have adopted and modified is a kind of algorithm which we can use it for the homogeneous systems.

Among of many multiprocessor system architectures, there are some systems will require each task running inside must be finished at a fixed time stamp, they are called real-time systems. As the previous definitions of T and P, let us consider on the case when the tasks have been scheduled by a scheduling algorithm, task t_1 and t_2 may be scheduled into the same processor p_1, and t_3 is assigned into the processor p_2 after a certain time delay waiting for t_1, processor p_2 will be triggered to deal with t_3 unconditionally. In the situation that t_1 does not hit the finished time as it was estimated, the image t_3 will be lying under the image t_1. To solve this problem, we might broadcast a signal to inform other processors that t_1 has just finished and it is ready to start its children tasks. Although the problem is solved, this system cannot be known as a real-time system anymore. In this study, our proposed algorithm is for both real-times and non-real-time systems. However, the implemented sample system is only suitable for running on non-real-time mechanism, because we cannot estimate the execution time for each task very accurately. In this study, we will propose a new modified version of HEFT to reduce the execution time for an entire DAG. On the other hand, we also describe our current system architecture as well as the promotional parts which should be additionally paid a little bit effort, in order to have more processors running synchronously.

2. **Motivation.** Working on the task scheduling we have to take account of the trade-off between minimizing communication costs and maximizing the concurrency of the tasks. In other words, besides we try to dispatch the tasks into some computation units to shorten the total execution time as much as possible, we also have to take care of the overhead increase by communication between each computation unit. We used to call the

total time that all the tasks in DAG have just finished as *make-span* [3]. Some algorithms had tried to shorten the *make-span* by merge up the tasks into a cluster solving on the same processor [4], or make a duplicate copy of t_i on the processors which will use the output of t_i as an input to handle the immediate children of t_i [5, 6, 7], that is quite useful to reduce the communication cost of the heterogeneous systems.

The scheduling algorithms can be classified into many different types based on the behavior that how to schedule each task to the processor. The well-known methods are list-based scheduling [7, 8, 9, 10], clustering based scheduling [4, 11], genetic-based scheduling [12, 13, 14].and duplication-based scheduling, although most of the recent studies have the trends combining different technique together. Run-time scheduling algorithms will consider the situation that the unscheduled programs are triggered while the processors are busy serving for some running tasks. In this study, our algorithm will not discuss about this issue but only propose a solution for the static tasks scheduling, and hence, the processors of these systems are only responsible for preprocessing the well scheduled tasks. Today, the scheduling algorithms become more and more powerful. Exploiting the advantage of the old algorithm and mending the fault almost are the trends of the current research jobs on this field, because we are almost reaching to the bottleneck of the optimal solution. The studies in the recent years are focus on the topic how to improve the existing methods by combining the different scheduling techniques [11, 15], such as insertion, duplication techniques are used in combination with the fundamental algorithms.

We have studied many articles to find out the most suitable solution for our systems, and finally, we have found that HEFT is a fairly good algorithm by now, it does not require too many computation resources to finish the jobs, and bring out the near optimal solution. However, when we try to assign the properties of homogeneous system to HEFT, we found that it does not work well as we hope for any cases. Again and again, we have tried many different methodologies to refine this algorithm in order to inherit all its advantages and to ride out the disadvantages as well. While we were searching for the algorithms, we found that there are few studies talking about how to map an algorithm from theory to a real implementation case, especially for the vector graphical parallel processing. And hence, we think that it remains a big challenge to do so, and we also hope this study result can help somebody who wants to solve the similar problems. In this section, we will go through to describe about OpenVG standard and it's opensource API [16]. However, to strengthen the basic knowledge of 2D vector graphics, we will waste a little time to talk about the current trend, advantage and disadvantage of them.

3. **Background Knowledge.**

3.1. **Dependence Analysis.**

There are two kinds of tasks dependence: *data dependence* and *control dependence*. Data dependence consists in two or more program parts or tasks when they have data transfer. For instance, there is an equation shown as below:

$$y = 2x^2 + 3x + 3.$$

The above equation is an obvious example for this constraint. According to mathematical principles in this equation, we recognize that we have to wait for the value of each term on the right hand side of equal sign, if we want to calculate the y value. For the term $2x^2$ at right of equal sign, we calculate the power of x, multiplying the result by 2, and then assign to a. After that, we calculate $3x$ and assign to b, and finally, accumulate two previous terms with 3 to get the value y. If we map this executed order into a topology graph, a DAG will look like in Fig. 1. In the above example, at first, we would easily realize that a and b terms are dependent on x, and they must wait for the value of x to be

ready before they can be calculated, because both of them are *immediate children* of x, while x is an immediate parent of a and b, as well as a predecessor of y , denoted *pred(y)*, whereas y is viewed as a successor of x, denoted *succ(x)*. In other words, y is the last node which have to wait for all three terms a, b and number 3 to be ready in memory, because y is a *leaf node* of this graph (enter and exit are two pseudo nodes with zero weight), it does not dominate anyone else. For each node without any parent in the graphs or their parents have been scheduled into the processors, we call them as *free nodes*, because they are free to be selected and put at any task queues after their parents' finished time.

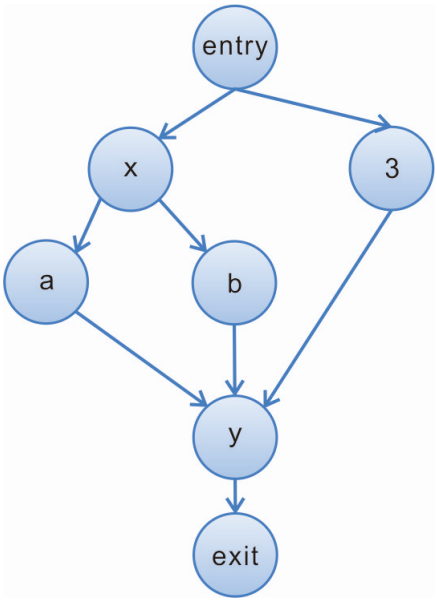

FIGURE 1. Data dependence relation

Comparing with the data dependence, control dependence does not cause by the data transfer among the tasks but depend on the data structure of a program. It is not easy to be recognized in scheduling, and the following pseudo code of Fig. 2 is a typical case:

```
1:   if random() % 10 < 5 then
2:     call loop_5_times
3:   else
4:     call loop_10000_times
5:   end if
```

FIGURE 2. Control dependence relation

In the above section of pseudo code, we cannot determine that what will do before line 1 is executed. The result of line 1 may cause a shorter delay which only has five turns of loop, or the longer procedure which will be involved to consume many processing resources. From the above sample, we have understood that might have many affective conditions while the programs are running. Random procedure, users' inputs, system interrupts etc., will directly affect to the decision of the next step in a running program. And hence, that will increase the complexity of the scheduling problem so much. Usually, two kinds of data dependence that we have mentioned above can be fractionized into the code level, and they are able to be solved by the complier reorganization technique at preprocessing step.

3.2. **Timing Constraint.** Real-time systems are usually classified into two types according to the level of timing constraint. If a system requires all tasks must meet the deadline or will be thoroughly fail, because it needs a very high accuracy on its jobs, and hence, it is called a *hard real-time* system. Such systems can be found at the production line which equipped with the robotic arms or some automatic control systems. However, in the soft real-time system, it is acceptable for the deadlines of jobs are missed. The follow-up tasks can be postponed as far as the predecessor tasks have finished, because the timing constraint is just a goal for the maximum performance. These systems usually occur around us in frequently, for example, when an online TV program receiver lags of receiving some data packages, it will not make any significant errors but only performs the frames in a little bit worse.

3.3. **System Model.** The system model of distributed multiprocessor systems usually has one global scheduler (or dispatcher) which takes charge of scheduling the tasks into each subsystem. Each subsystem may also have its own local scheduler for receiving the arriving tasks, and this local scheduler is responsible for rescheduling the execution order of the tasks and assigning them to the local processors. Sometime this scheduler even rejects the tasks if the local processors are too busy. And when a task is finished, it will send back the result to global scheduler. This system model might look like the Fig. 3 has shown below:

Another type of multiprocessor system is a stand-alone multi-core system. It looks like a miniature of the systems described above. Usually, the scheduler is a general purpose processor (GPP) running for dependence analysis, besides this, it also takes the management of tasks scheduling or even the responding to users' requirements. Major different from the first system model of this one is the connected media, every processors in this model do not communicate through a network but pass the data by a high speed BUS or shared memory as we depict at Fig. 4. The difference which affects to the scheduling algorithms of these two system models is the first one may need consideration on the communication cost between each subsystem whereas the second one might omit it.

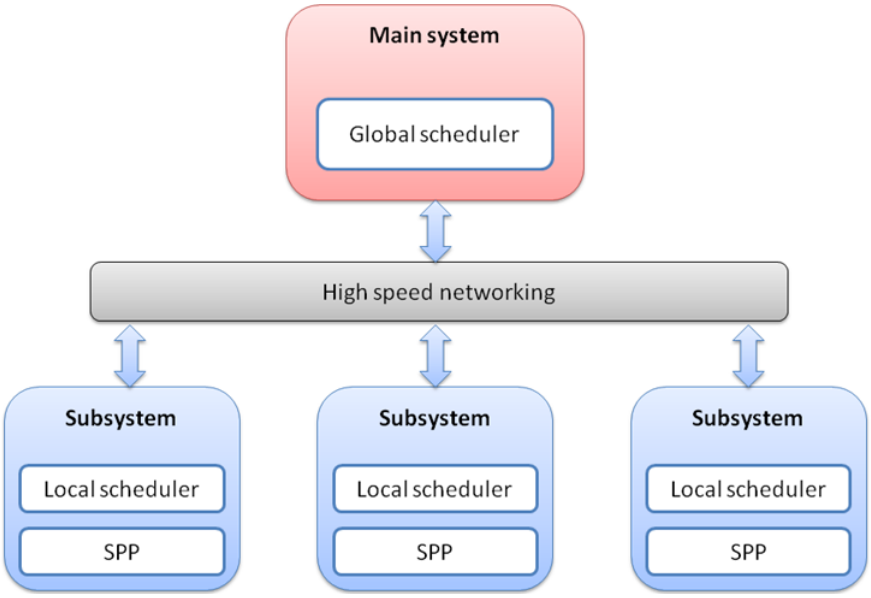

FIGURE 3. Distributed multiprocessor system model

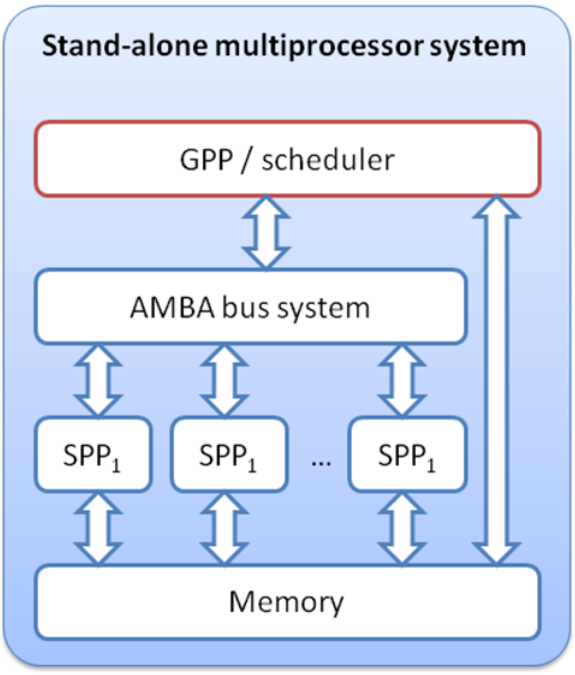

FIGURE 4. Stand-alone multiprocessor system model

3.4. **Critical Path.** For any DAG, each node will be connected by at least one edge, the number of edge pointing to a node is called *in degree* of that node, and the number of outgoing edge from a node is called *out degree* of that node. The edges and nodes connect together to form a path, and the longest path starts from entry node and ends at exit node is called the *critical path*, the *longest* condition does not come from the number of edge but define by the computation time of the nodes and the communication time of the edges on that path. In some situations, critical path may not be unique in DAG. Two or more critical paths having the same length can be formed at the same time by more than one different node. Another attribute of critical path is that it always starts at an entry node and stops at the exit node of DAG. To proof this truth, let $\mathbf{G} = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph with $\mathbf{V}$ is a vertex set and $\mathbf{E}$ is edge set in $\mathbf{G}$, w is computation cost of $\mathbf{V}$ and c is communication cost of $\mathbf{E}$, we can easily use the contradiction proof method:

If the first node of critical path (cp) which denote by n_1 is not an entry node of $\mathbf{G}$, then it must have a n_0 belong to predecessor set of n_1, and hence, there is a edge $e_01 \in \mathbf{E}$, and the new length of critical path q denote by $length(q)$ should be:

$$length(q) = w(n_0) + c(e_01). \tag{1}$$

When $w(n_0) > 0$, it imply that $length(q) > length(cp) - a$ contradiction, likewise for the exit node so that cp is start in entry node and end in exit node of G.

3.5. **Node Levels and Scheduling Priority.** To determine which node is the most important in each scheduling turn, we have to calculate the node levels that are usually defined by one of two parts: *top levels* (tlevel) or *bottom levels* (blevel). To calculate the tlevel of n_x, again, let us assume $n_x \in G$, thus we have:

$$tlevel(n_x) = \max_{n_i \in pred(n_x)} \{tlevel(n_i) + w(n_i) + c(e_{i,x})\}. \tag{2}$$

Equation (2) implies that the top level (tlevel) of n_x is the longest path from the entry node (n_entry) to nx excluding $w(n_x)$. Opposite to top level, bottom level of n_x is defined as equation (3):

$$blevel(n_x) = \max_{n_i \in succ(n_x)} \{blevel(n_i) + c(e_{x,i})\} + w(n_i). \tag{3}$$

It is very clear, bottom level has defined the longest path from n_x (include n_x itself) to exit node n_exit. For the easier illustration, we depict Fig. 5 as shown below. Summation of the pink nodes and edges which start from n_x to n_exit and follow the downward direction is the $blevel(n_x)$, and the opposite direction including the last blue arrow on the top of n_x is $tlevel(n_x)$.

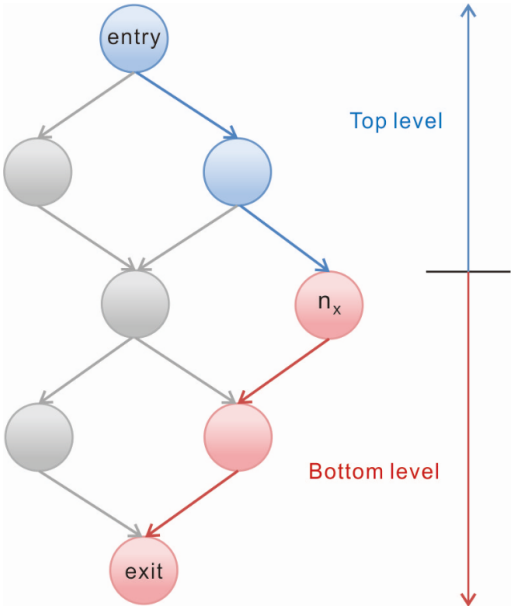

FIGURE 5. Top level and bottom level

The algorithm to determine the rank of each node is either using tlevel or blevel, for the blevel, it recursively bottom up summate the $w(n_i)$ and $c(e_{x,i})$ from n_exit to n_x, again, if the system architecture is as Fig. 4, it remains the term $w(n_i)$ in each recursion.

4. Methods.

4.1. **HEFT Algorithm.** First of all, HEFT is one of the list-based scheduling algorithms, because the characteristic of them is creating a priority list at the first step. According to a sorted priority list, HEFT will assign each task to a suitable processor such that the task can be finished as soon as possible. The following pseudo code is quoted from the original paper of HEFT. From Fig. 6, we know that HEFT will recursively try to search for local optimal in order to finally have the global optimal, and the time complexity of HEFT is $O(v^2 \times p)$.

4.2. **Problem in HEFT.** If we assume that the processor set in homogeneous system is, and the nodes of Fig. 7 will be sorted according to their levels as the order showing in this set $V = \{v_1, v_5, v_6, v_7, v_3, v_2, v_4, v_9, v_8, v_{10}\}$. *Entry* and *exit* are pseudo nodes with zero weight, so they will be ordered in the first and the last positions of V respectively (they have been omitted here). The next step of HEFT algorithm will schedule each node in V to the processors of P, and then we will get the result as shown in Fig. 8.

1: Compute $rank_u$ for all nodes by traversing graph upward, starting from the exit node.

2: Sort the nodes in a list by nonincreasing order of $rank_u$ values.

3: **while** there are unscheduled nodes in the list **do**

4: **begin**

5: Select the first task n_i in the list and remove it.

6: Assign the task n_i to the processor p_j that minimizes the (EFT) value of n_i.

7: **end**

FIGURE 6. The HEFT algorithm

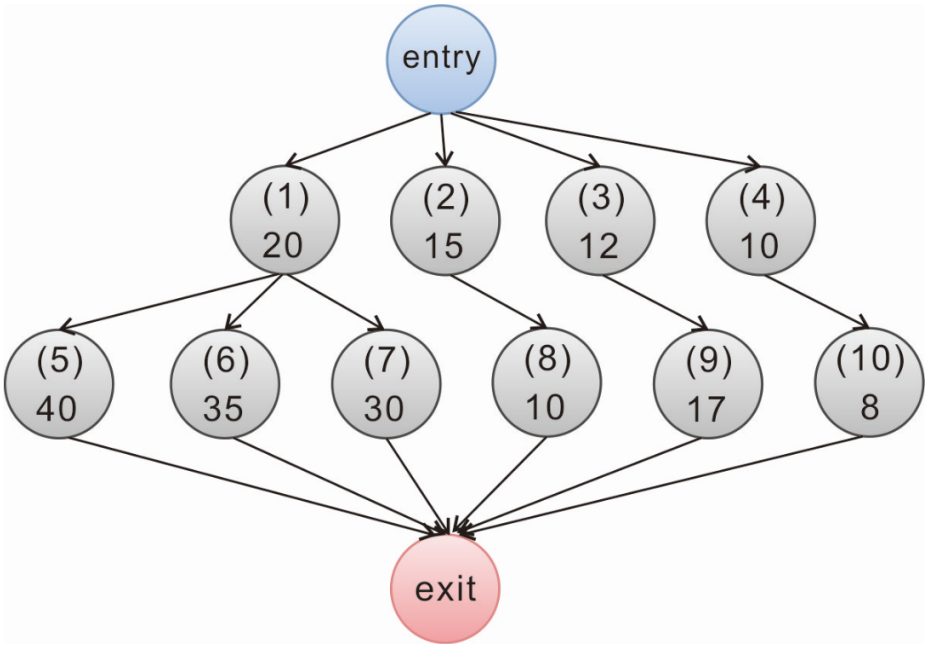

FIGURE 7. Sample DAG

At first, v1 is scheduled in p_1 because all processors are idle at beginning and have the same computation capacity, after that, v_5, v_6 and v_7 are scheduled into p_1, p_2 and p_3 respectively because they are dependent on v_1 as we have seen in Fig. 7, it means that v_5, v_6 and v_7 cannot start until v_1 have finished, so they can only be scheduled later than 20^{th} time unit, after v_6 and v_7 are scheduled, they will form two holes at two processors from the earlier time before they can start, we used to call such a hole as *timeslot*. After v_7 is scheduled, v_3 is the highest priority among the remaining nodes in V, so it is chosen to assign to the earliest start time among three processors. Because HEFT uses the insertion-based policy and bases on the insertion condition to determine whether the nodes v_i can be inserted to the earlier time timeslot at processor p_j, it will satisfy if the subtractive result from start time (ST) of scheduled task v_{n+1} in processor p_j to finish time (FT) of scheduled task v_n on processor p_j is greater than or equal to the computation cost w_i on p_j. This condition is described as equation (4):

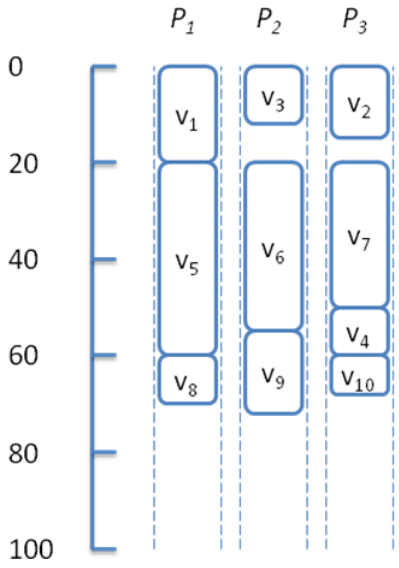

FIGURE 8. Scheduling result of HEFT

$$ST(v_{n+1}, p_j) - FT(v_n, p_j) \geq w_i. \tag{4}$$

In this case, v_3 holds the above condition and p_2 is the most suitable candidate. The remaining nodes are scheduled into each processor respectively basing on earliest start time and insertion policy. The final make-span is dominated by p_2 of the sequence v_3, v_6 and v_9 which spends 72 time units. Because of the unbalance of Fig. 7, it yields an idle timeslot of p_2 and p_3 in the first 20^{th} time unit, however, this idle time usually cannot be filled up by the later nodes. The waste of the computing resources in this situation might become huge as the graph grows larger. The newer versions of HEFT have improved a lot of the performance. Those new algorithms might use the duplicate technique to reduce the communication time in the entire scheduling [6, 7, 17], and yet, we are focus on the algorithm which will be used for homogeneous system and without any communication cost, so this technique is not considered in our proposed algorithm.

4.3. **Improved Heterogeneous Earliest Finish Time (IHEFT).** The IHEFT is considering on the phase modifying the original HEFT for using in homogeneous systems instead of the heterogeneous system, , and hence, some of the mechanisms still inherit from HEFT, the following sections will describe some definitions of our algorithm.

4.3.1. *Definition of IHEFT.* Tasks are ordered in our algorithm using the upward ranking which is defined recursively as below:

$$rank(v_i) = w_i + \max_{v_j \in succ(v_i)} \{blevel(v_j)\}. \tag{5}$$

That is why the sorted result of set V of Fig. 7 presents so. And the earliest finish time (EFT) and earliest start time (EST) definitions are adopted from HEFT and they are modified to be suitable to use in homogeneous systems as (6) and (7).

$$EST(v_i, p_j) = \max\{Available_{P[j]}, \max_{v_m \in pred(v_i)} (EFT(v_m, p_k)). \tag{6}$$

$$EFT(v_m, p_k) = w_{i,j} + EST(v_i, p_j). \tag{7}$$

Where $Available_P[j]$ is the earliest time which processor p_j is available for task execution, and $pred(v_i)$ is the set of immediate predecessors of task vi. The inner max block in the EST equation returns the time when all immediate predecessors of v_i have finished, and the outer max block will choose the latest time by comparing the available time on p_j with the latest predecessor finished time, because both of these constraints must be satisfied to be able to put task v_i on p_j.

After ranking and sorting the tasks of DAG into list V, the most upper task in list is the highest priority one, it will be popped out and scheduled to the most suitable queue $q_i \in Q$ of processor $p_j \in P$ where $|Q| = |P|$ such that the earliest start (EST) condition is satisfied. The major different point of our algorithm from HEFT is not only checking the idle timeslots form by the earlier scheduled nodes in q_j and then using the insertion-based policy to insert vi but also reschedule the later position scheduled nodes in q_j if

$$0 < ST(v_{n+1}, p_j) - FT(v_n, p_j) < w_i. \tag{8}$$

And hence, the insertion condition is not fixed for the timeslots which are larger than or equal to $w(v_i)$ in our algorithm. Furthermore, it will continue trying to find the $EST(v_i, p_j)$ for each p_j although it has already found a timeslot at previous processors. To initial the start time (ST) of v_i, it is defined as (9) below:

$$ST(v_i) = \max\{mpkespan(P)\} + 1. \tag{9}$$

On the general occasion, both $ST(v_i)$ and $ST(v_i, p_j)$ have the same meaning, they denote the start time of v_i, but the first one does not care about what processor p_j is. To search the timeslots, we start scanning from the earliest scheduled task in q_j, whenever a timeslot is found or in the case we do not found any timeslot and have to put v_i at the last position of q_j, it will record and compare the current EST with the current ST of v_i to determine which position will be the best for performance.

$$ST(v_i) = \min\{ST(v_i), EST(v_i, p_j)\}. \tag{10}$$

Equation (10) is used to determine the final ST of v_i. To record the ST of v_i, we have to store the both index of that temporary EST as well as the processor identity for comparing during the searching procedure and the final inserted condition.

4.3.2. *Procedure of IHEFT.* The procedure of IHEFT is shown below in Fig. 9.

4.3.3. *Example for explanation.* To be more clearly describing the procedure of our proposed algorithm, we will use Fig. 7 as a sample, and the number of processor in set P is three. After finish the first two steps which all the list-based scheduling algorithm might do, we obtain a set of descending sorted tasks according to ranking values in list V as shown below in Table 1. The subscript numbers at V elements are the weight of the nodes.

At step 5 of Fig. 9, each task will be selected from the first of V, it is checking EST with v_k on each processor p_j at step 8, the earliest start time of current v_i will be reconsidered at step 9 if the conditions are satisfied. After all p_j have been checked, in the case there is no timeslot found at step 8 and 9, vi will be considered to put at the tail of q_j at step 12. At step 14, we insert v_i into one list of Q, at the same time, the position of v_i in V should be updated. And then we have to make sure that whether v_i is inserted into a

```
1:  compute the rank of each node in DAG by traversing graph upward (blevel)
2:  sort the ranking result of the nodes by decreasing order in list V
3:  while there are unscheduled nodes in the list V do
4:  begin
5:  select each vi from the top of list V
6:  for each pj in processor set P do
7:  for each element vk in pj do
8:  if timeslot exists at FT(v_k, p_j) of qj and FT(v_k, p_j) ≥ EST(v_i, p_j)
9:  set ST as min{FT(v_k, p_j), ST(v_i)}
10: end if
11: end
12: set ST as min{EST(v_i, q_j), ST(v_i)}
13: end
14:   insert vi to ST and update the position vi at V
15:   if vi is inserted into a time slot and wi > timeslot > 0
16:     reschedule all tasks in P later than vi
17:   end if
18: end
```

FIGURE 9. IHEFT scheduling algorithm

TABLE 1. (a) Ranking value of each node and (b) Sorted nodes

Task	Weight	Rank	List V
1	20	60	1_{20}
2	15	25	5_{40}
3	12	29	6_{35}
4	10	18	7_{30}
5	40	40	3_{12}
6	35	35	2_{15}
7	30	30	4_{10}
8	10	10	9_{17}
9	17	17	8_{10}
10	8	8	10_{8}

timeslot and satisfy the condition at step 15 and reschedule all the nodes later than v_i. For example, after tasks v_5, v_6 and v_7 have been scheduled, two timeslots appear at the earlier time slice on p_2 and p_3, and when v_3 and v_2 are inserted to p_2 and p_3 respectively, two timeslots become smaller, they do not affect the start time of later nodes v_6 and v_7, so we only rearrange list V as this order $v = \{v_1, v_5, v_3, v_6, v_2, v_7, v_4, v_9, v_8, v_{10}\}$. And when the next element v_4 is inserted into p_2, it will postpone the start time of v_6, and hence, we have to reorder list V for a new rescheduling. In any case, we cannot change the topology of the DAG after reorder list V, it is always equivalent to the original one, and hence after task v_4 is scheduled, we have $v = \{v_1, v_5, v_3, v_6, v_2, v_7, v_4, v_9, v_8, v_{10}\}$, the procedure will not stop until all the nodes have been scheduled. We have the final result which is shown at Fig. 10, and the schedule steps are listed in Table 2. Make-span of this result only spend 69 time units for entire DAG, it is fewer than HEFT algorithm 3 time units, and the total saving time from all processors is 13 time units. The make-span of this sample is formed by the v_1, v_5 and v_{10} at q_1.

In the above table, we have some symbols for the actions described below:

TABLE 2. Scheduling steps of sample DAG

Step	Node	Action	Result	CPU1	CPU2	CPU3
0	-	-	1 5 6 7 3 2 4 9 8 10	-	-	-
1	1	A	**1** 5 6 7 3 2 4 9 8 10	**1**	-	-
2	5	A	1 **5** 6 7 3 2 4 9 8 10	1 **5**	-	-
4	6	A	1 5 **6** 7 3 2 4 9 8 10	1 5	**6**	-
5	7	A	1 5 6 **7** 3 2 4 9 8 10	1 5	6	**7**
6	3	I	1 5 **3** 6 7 2 4 9 8 10	1 5	**3** 6	7
7	2	I	1 5 3 6 **2** 7 4 9 8 10	1 5	3 6	**2** 7
8	4	I	1 5 3 **4** 6 2 7 9 8 10	1 5	3 **4** 6	2 7
9	-	R	**1 5 3 4 6 2 7** 9 8 10	**1 5**	**3 6**	**4 2 7**
10	9	I	1 5 3 4 **9** 6 2 7 8 10	1 5	3 **9** 6	4 2 7
11	-	R	**1 5 3 4 9 6** 2 7 8 10	**1 5**	**3 9**	**4 6**
12	2	I	1 5 3 4 9 **2** 6 7 8 10	1 5	3 9	4 **2** 6
13	-	R	**1 5 3 4 9 2 6** 7 8 10	**1 5**	**3 9**	**4 2 6**
14	7	A	1 5 3 4 9 2 6 **7** 8 10	1 5	3 9 **7**	4 2 6
15	8	A	1 5 3 4 9 2 6 7 **8** 10	1 5	3 9 7 **8**	4 2 6
16	10	A	1 5 3 4 9 2 6 7 8 **10**	1 5 bf 10	3 9 7 8	4 2 6

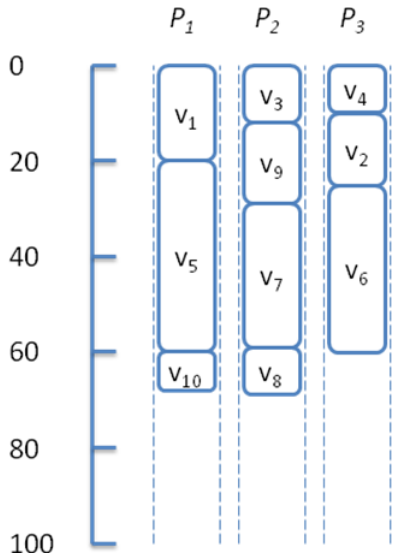

FIGURE 10. IHEFT scheduling result

- A: append a task to the tail of one queue list
- I: insert a task into a timeslot of one queue list
- I: insert a task into a timeslot which is smaller than the weight of this task. Right after this, it will yield a reschedule action immediately.
- R: reschedule action

5. **Implementation.**

5.1. **System Architecture.** Our system is designed as Fig. 11 shown below. According to the processing procedure, we separate this system into two parts *hardware* and *software* models, the software part is responsible for allocating the resources, preparing the tasks to the memory and also working for some graphical processing. In our later

assumption for the algorithm using, all graphical processing is fully support by hardware, software only works as a scheduler. Our system is constructed by one ARM9 GPP and one FPGA accelerated graphical processing circuit. The GPP is the computation resource for software, and the FPGA circuit is for hardware. In general, the processing speeds of hardware are much faster than software, and hence, software usually prepares amount of tasks at once and send to hardware to make more time in the next turn of preparing the jobs. After receiving the tasks, hardware starts to solve them one by one, in this duration, the hardware cannot be interrupted. And hence, the execution time of these tasks should be estimated as accurate as possible. To reduce the developed time, flash player is modified from an open source project Gnash, and the aim of this part is for converting the swift binary instruction to OpenVG commands.

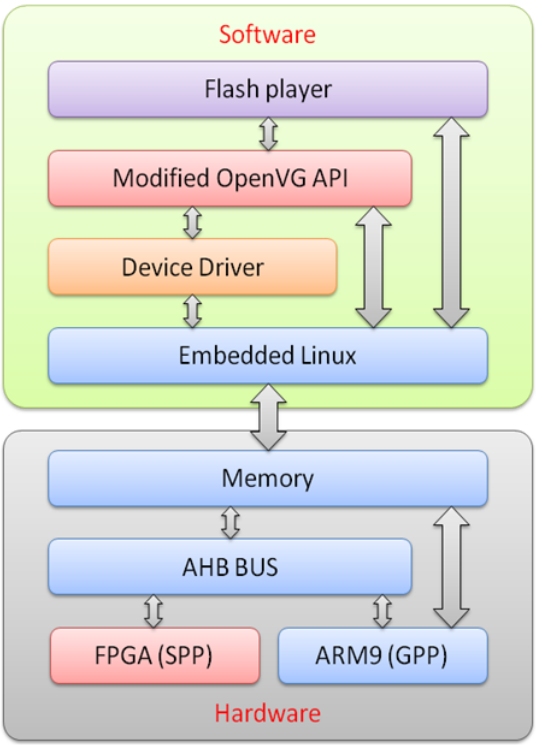

FIGURE 11. Software and hardware architecture

5.2. **Hardware Developed Environment.** The following image is fetch from the Socle CDK training document, it depict the detail function of each component on the board. This platform has two cores, one ARM9 GPP and one programmable FPGA, they communicate through the AHB BUS by storing the information at some specify areas in memory. The frequency of FPGA core is fixed at 40MHz, and the frequency of ARM9 has to be slower than FPGA if we intend to have they work synchronously. The memory provides by this platform is about 64MB SDRAM and 16MB static flash memory. TFT LCD is a 3.5 inch (320x240) touch screen; however, this board also provides a PCI slot for plugging the external VGA card, so it is a solution for testing the high solution pattern.

5.3. **Modified OpenVG API.** The first step we did after fetching the OpenVG API from the internet is trying to modify it into two types of library, Non-OS-based and OS-based library, because many constraint such as memory allocating, system call, timing, interrupt and job preemption can be neglected under the Non-OS-based environment, it becomes very simple that the tasks are scheduled as an exclusive sequence working on

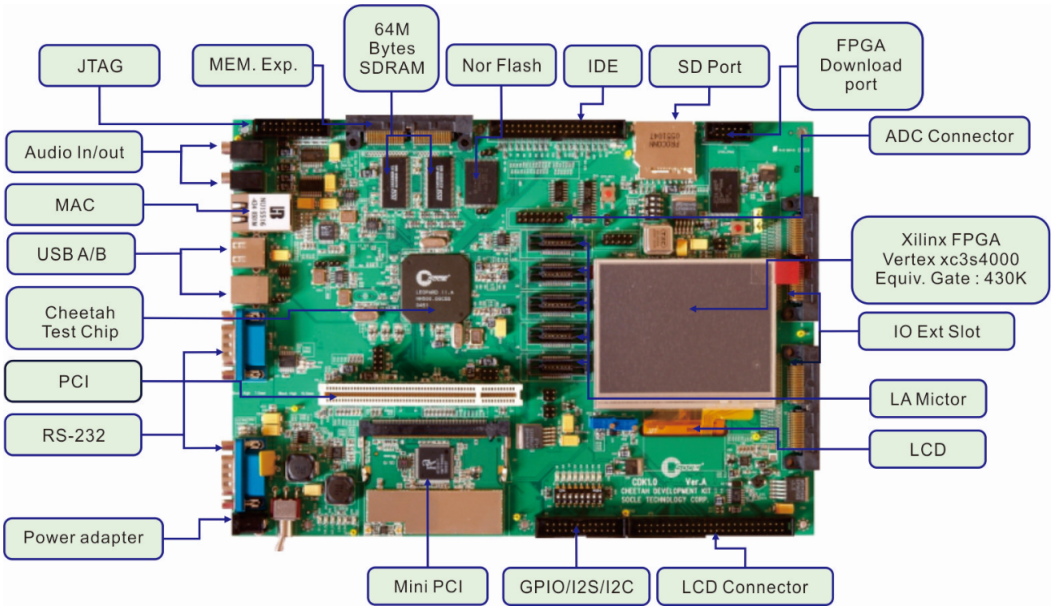

FIGURE 12. Socle CDK development board

their own processor, and hence, we are willing to waste some time on developing this part for debugging purpose although it cannot be combined into final release. The simulation environment to launch the Non-OS library is on CodeWarrior for ARM developer suite v1.2, and the hardware platform is CDK development board.

The second step we have to prepare for modifying the API is trying to rebuild (porting) it into the Unix-based environment, because we will use the Unix-based API in our final system, this is a crucial step of our project. Because the original API is written for Macintosh and Microsoft Windows OS, and to make it work on Unix-based OS, the main job in this step is reviewing all the codes of original API and totally replacing all ineligible system dependent commands by the suitable one on Unix-based-OS. The result of this step is presented in Fig. 13 a), by the way, Fig. 13 b) is the result drawing by hardware after many efforts have made. This sample tiger is drawn by 305 paths providing by Khronos as a bench mark for demonstration or a guideline for debugging. And the third step of this part last long through the time for developing this system. The hardware developers always try to save the resource of that circuit and make it work faster, so they will get rid of some burdensome functions from that chip, and hence to coordinate with the hardware, we have to solve those problems by software. In addition, this modified API (MAPI) also needs to determine the processing flow for each coming task, it has to check that whether the current task could be scheduled on hardware or it should be redirected to the GPP.

5.4. **Double Link-List and Memory Allocation.** For load balancing of hardware and software, our system is designed for running on a double-buffer scheduling list. As we have known, the GPP works as a scheduler in this system, so it has to pass the data to the SPP whenever a task has to accelerate by hardware. This scheduler might fall into an idle status after it has passed the first piece of data to the RAM and triggered the GPP. And hence, to avoid being waste the resources of GPP while the SPP is busy, the page flipping (ping-pong buffering) scheme is using in our system. In this mechanism, there are two chunks of memory have the same size, the active one is usually called the front buffer and the other one is back buffer, they usually serve the different target at the same time

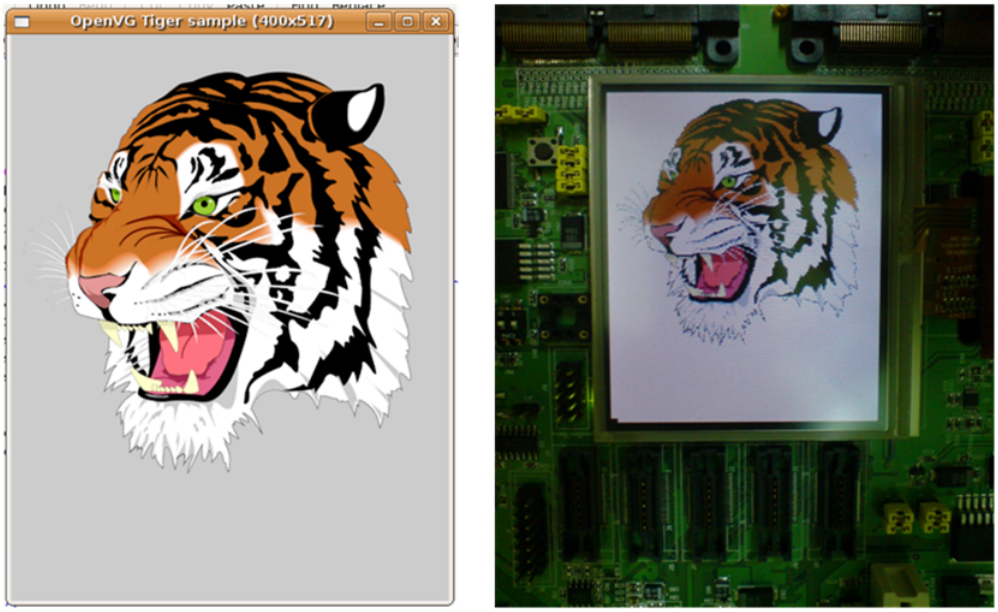

FIGURE 13. Modified OpenVG API implementation result

although they might have the same purpose or even the same content. There are several masses of memory must be allocated right after the driver is loaded. The double link-list buffer as well as the other storing spaces is all handled by the OpenVG MAPI, there are some safety checking mechanism and restriction to prevent the users try to allocate the memory or modify the other program or system data illegally. In the current version, the memory are mapping to four primary areas as Fig. 14 which highlight with different border color, and the size of this area is depicted on the graph, most of them have a very complex data structure, whereas some are simply reserved for the hardware temporary buffering purpose, so they do not need to do anything but leave a start physical address at the specific memory location for the hardware.

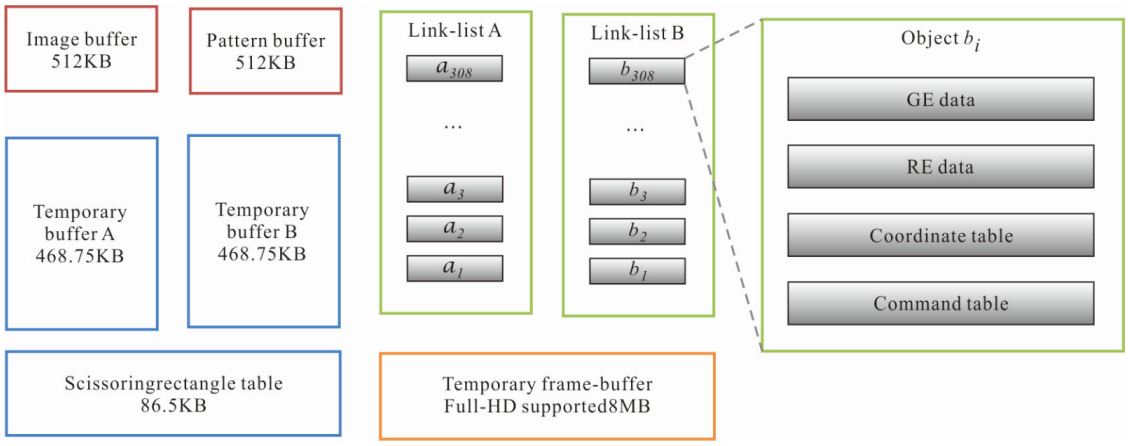

FIGURE 14. Memory mapping

5.5. **System Flow.** To let the system works smoothly between hardware and software, we must have a reasonable process arrangement. The following flow chart will describe

the current single GPP and SPP system hardware and software workflow. There are two situations that the hardware might be triggered, the first one is when the task arrives and the other one is when all tasks are scheduled into link-list and the program has reached the end of it. The step with a star symbol (*) in Fig. 15 only require polling an inactive link-list. Our system is developed under the Unix-based embedded system, to be safely use the memory resource, prevent the illegal accessing to system and let the software easily communicate with hardware, we must have a device driver that provides an simply interface and also takes charge of these sensitive issues of security [18].

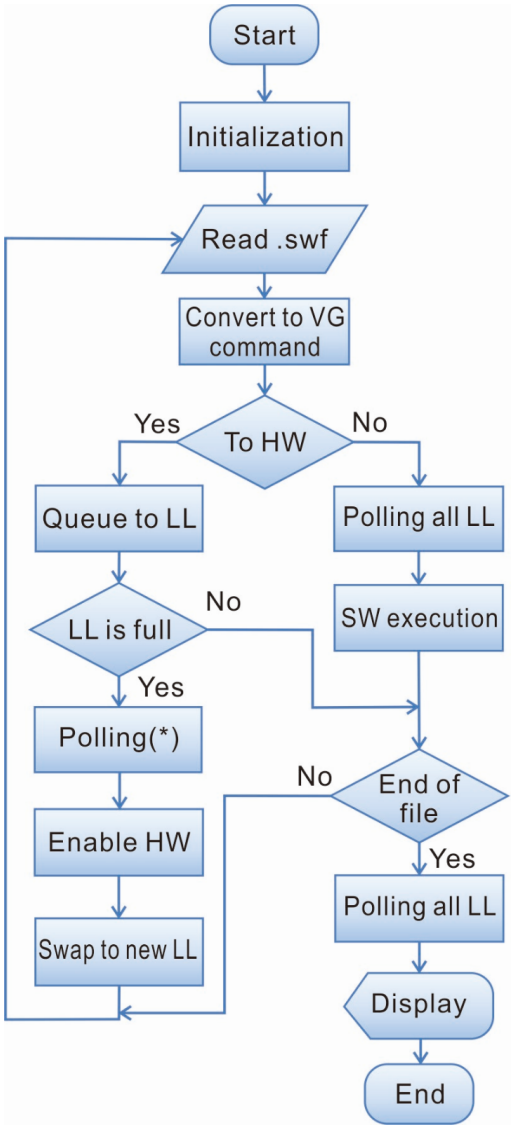

FIGURE 15. System workflow

5.6. **The Multi SPP Model.** As the current system architecture, to duplicate the more homogeneous processors, it will need the more memories for each additional processor. However, because we are using the link-list technique, on the other hand, the evaluation of the execution time cannot be very accurate for all cases, the tasks might be finished sooner or later than the estimated execution time, and hence, it is required to have a synchronize mechanism for these individual SPPs to let them follow the preceding constraint of the tasks. We should have a counter for each task and when the task is scheduled to its

processor, a positive initial value which represents the number of its immediate parents is set. The counter will be countdown 1 whenever its immediate parents have done or it will be verified to ensure that the value inside is zero before getting start the task. For easier understanding, we only depict the relevant components with two SPPs in Fig. 16. Two rectangles with label LL are the link-list for storing the paths information similar to the single SPP version. The little change is only adding a serial identifier number to a task and two extra pointers, one is pointing to a location where it stores a set of immediate parents' identifier of that task, and the other one is to the start address of an array storing a counter. The purple rectangle (TS) is prepared for counters and immediate parents' identifiers. The above system architecture is recommended for implementation of a multiprocessor vector graphical processing system. With that architecture, we will easier to use the scheduling algorithm for parallel processing. In the next chapter, we will come back to evaluate our proposed algorithm.

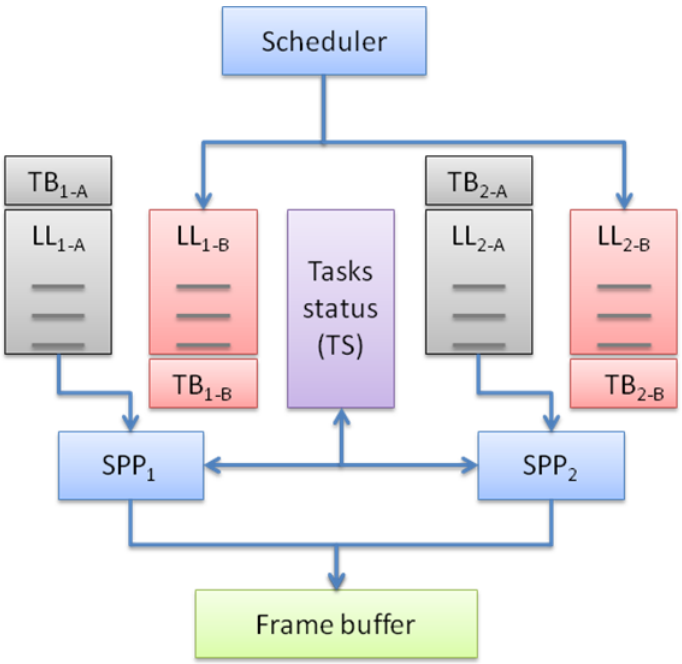

FIGURE 16. Multiprocessor system model

6. **Experimental Results and Discussions.** We will address two experimental results focusing on the different factors that will affect to the performance of our algorithm. These two results are from comparing with two algorithms HEFT and the later version Look-ahead HEFT [1, 19].

6.1. **Experimental Environment.** For the convenience of validating our algorithm, we have implemented a tool which provides an interface can freely change between the different algorithms. This tool includes three parts: one setting panel, one graph drawer as well as a timing display panel as shown in Fig. 17, 18, and 19, respectively.

The options will widely affect to the characteristic of the graph which are necessary for analyzing and comparing an algorithm in the various factors. They are simply summarized here:

Max. in degree: the number of incoming edge of each node, of course, they are generated less than or equal to this number, and connected randomly to the upper level nodes.

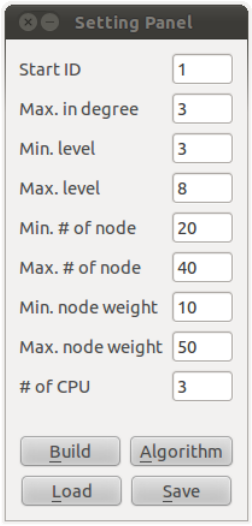

FIGURE 17. Setting panel for algorithm analyzing tool

Min./Max. level: is the level (depth) of the graphs, the users are able to make the graph looks tall and thin or short and fat according to the this option.
Min./Max. number of node: they specify the complexity of graphs, however, because the graphs generated randomly, sometime it might have less node than this minimum boundary.
Min./Max. node weight: this options can specify the scope of weight of the nodes, they can make the graph become unbalance with a wide range definition.
Two other parts of this tool are for displaying the results; they are organized as the images below:

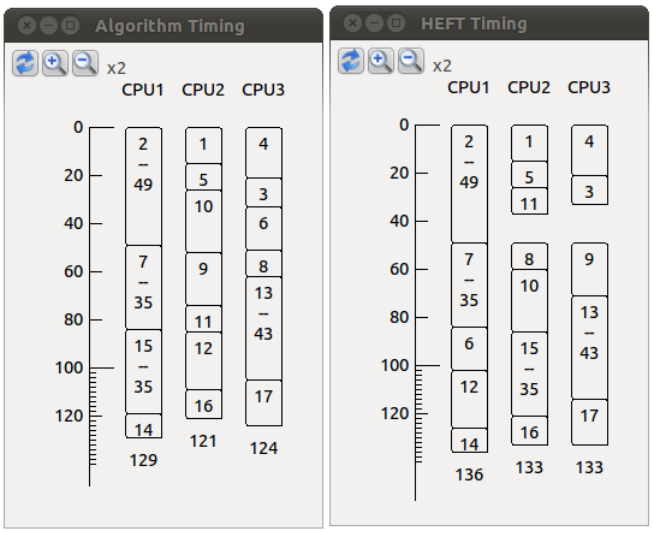

FIGURE 18. Timing comparison panel

For easier comparing the result, the timing panel fully provides the relevant information of the results from algorithm. The panel of graph drawer displays the graph architecture. It denotes the node ID and weight inside each node and connects each node by arrows as we see in Fig. 19.

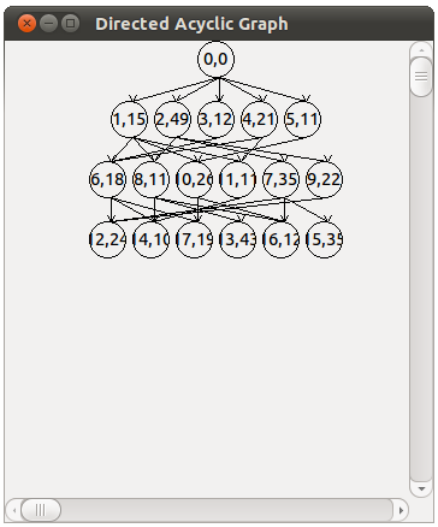

FIGURE 19. Graph drawer panel

6.2. **Experiment Steps.** We have implemented our algorithm as well as two others HEFT and Look-ahead HEFT separately, of course, the algorithms in these experiments are only using the scheduling procedure for the homogeneous systems; they do not include any attribute of heterogeneous systems.

In these experiments, we focused on modifying three properties of the graphs to present the behavior of our algorithm.

- The first subtest tried to understand how the increment of the number of nodes will affect to the scheduling performance, because the fat and short graphs might have more free nodes after each turn of scheduling than the longer, we were wondering whether the fat graphs will have the better condition for insertion, or they will have more chances to leave a timeslot after scheduling a high out degree node.
- The second subtest had tried to increase the level of graphs while other attributes were kept unmodified. With the same number of nodes, the taller graph might less parallelism capability than the short one, so what will happen with the proposed algorithm under this condition.
- And the last subtest had a larger architecture, both dimensions (level and number of nodes) are huge, and we had tried to solve with different number of processors.

Each turn of one subtest will randomly generate three hundred graphs according to the attributes which are specified by setting panel of our tool.

In addition, we should mention about how to compare the output value of each experiment. There are three comparing factors will be considered here:

- The times of win, lost and tie for each algorithm
- The percent of time that our algorithm has overcome the other
- The total percent of time that our algorithm has reduced from all processors

The first comparing factor had simply accumulated the turn of win, lose and tie for each algorithm. The more detail of second and third factors are presented later with the graph. The following Table 3 describes the setting of our algorithm compares with HEFT:

6.3. **Experiment Result of IHEFT and HEFT.** In Fig. 20, we found that the total times of win grows fast when the number of nodes increase and the cases of lose are lightly pulled down. The chance which let the case of win grow up is from reducing the case of tie. To determine the final result of win, tie or lose, we collected the maximum value

TABLE 3. Experiment setting of IHEFT - HEFT

Options	Subtest 1	Subtest 2	Subtest 3
Max. in degree	3	3	3
Min. level	3	**5 - 9**	10
Max. level	8	**9 - 13**	15
Min. number of node	**20 - 60**	70	400
Max. number of node	**50 - 90**	90	500
Min. node weight	10	100	10
Max. node weight	50	10000	50
Number of CPU	3	3	**3 - 16**

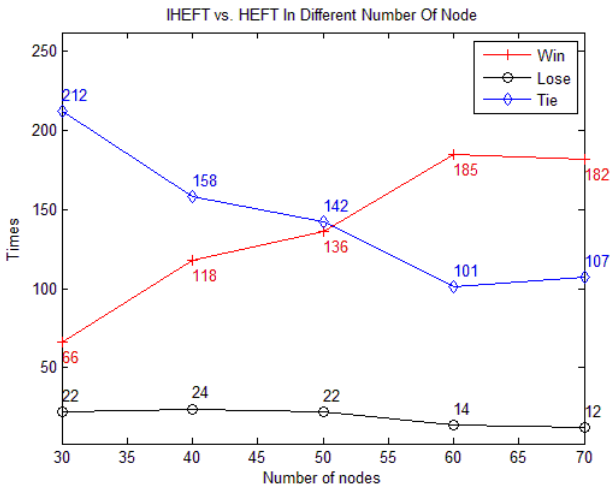

FIGURE 20. The times of win, lose or tie in different number of nodes

which were generated by three CPUs of both algorithms, and then compared them to get a conclusion. Only collecting the maximum value is a formal way to compare two algorithm performances in scheduling. To be more understandable about the performance of our algorithm, we also tried to evaluate the percent of time that our algorithm has saved against HEFT. In Fig. 21, we compared the average winning result from two algorithms based on this formula which is denoted by a blue line:

$$AverageSpeedup = \left(\frac{\Sigma_{n_i \in ALL} \max_{p_j \in P}\{FT_{HEFT}(p_j)\}}{\Sigma_{n_i \in ALL} \max_{p_j \in P}\{FT_{IHEFT}(p_j)\}} - 1 \right) \times 100, \tag{11}$$

Eq. (11) is defined for calculating the average speedup percentage of comparing IHEFT against other algorithms. In general, our algorithm performs better than HEFT in each number of nodes. In this formula, n_i is the turns running this experiment, FT is execution finished time of algorithm running on processor p_j, it includes the idle time between the first task and the last task reside in it. The red and black lines present the win and lose ratio of time, they are determined by the following two formulas.

$$Win = \left(\frac{\Sigma_{n_i \in WIN} \max_{p_j \in P}\{FT_{HEFT}(p_j)\}}{\Sigma_{n_i \in WIN} \max_{p_j \in P}\{FT_{IHEFT}(p_j)\}} - 1 \right) \times 100, \tag{12}$$

$$Lose = \left(1 - \frac{\Sigma_{n_i \in LOSE} \max_{p_j \in P}\{FT_{HEFT}(p_j)\}}{\Sigma_{n_i \in LOSE} \max_{p_j \in P}\{FT_{IHEFT}(p_j)\}}\right) \times 100, \tag{13}$$

By the side comparing all winning, losing or tie situation of blue line, the equation (12) and (13) show that we only compared the win or lose rate among the win or lose result in order to know how much of time they had got different separately.

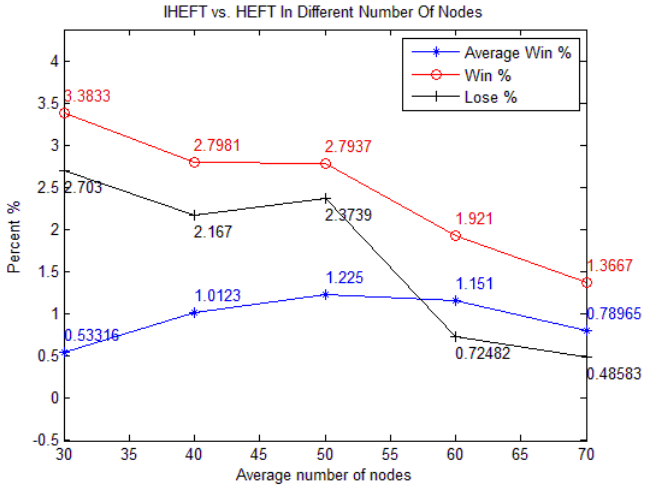

FIGURE 21. Time saving in different number of nodes

Finally, we also want to know the total time that our algorithm had saved against HEFT in this subtest, because the above evaluation is only presented the rate of time when the last processor has finished. In the real world, the sooner finished processors are also able to be scheduled to do something, so if an algorithm can make other sooner finished processors idle, it should be considered as an advantage of that algorithm. In Fig. 22, the blue line is adopted from Fig. 21, and the red line is defined by the following equation:

$$AverageSpeedup = \left(\frac{\Sigma_{n_i \in ALL}\{FT_{HEFT}(p_j)\}}{\Sigma_{n_i \in ALL}\{FT_{IHEFT}(p_j)\}} - 1\right) \times 100, \tag{14}$$

Summarizing of the subtest we found that our algorithm highly dependents on the shape of the DAG, too few or too many nodes with the same level (depth) of graphs might bring the performance down, that will approach to the performance of HEFT.

Our second subtest tried to know how the performance changes when we set the different level to the testing patterns. In this subtest, the evaluation formulas were defined as the first subset. And now, let us consider on the result depict here in Fig. 23, the win rate is still high although it gets lower at the end. The cases of lose lightly grow when the level is bigger, we believe that all lines will approach to each other because the higher level of DAG will makes the precedent constraint between each node grows as well. It might be no use to apply any parallel algorithm to those situations.

Again, in this subtest, we also observe to the rate of time saving comparing to HEFT in different level. The average saving time is about 1.63% lying between 1.33% and 1.8%. It seems quite unusual if we refer this result to the Fig. 23, because the performance of this result keeps growing up while the tail of the previous falls down, however, it is not impossible. There is not very much difference comparing between two total performance

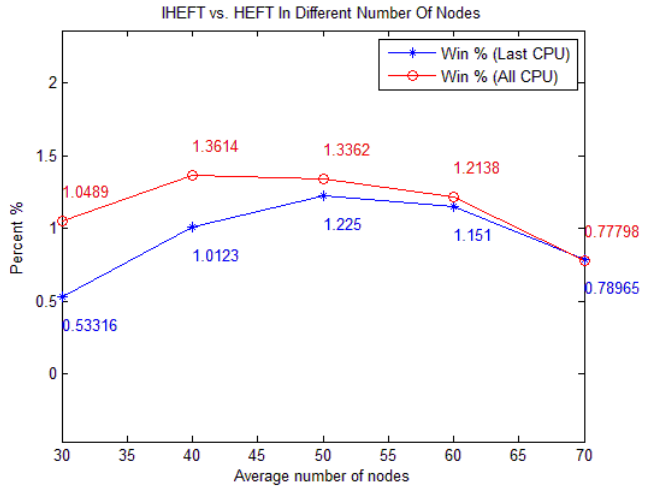

FIGURE 22. Total time saving of different number of nodes

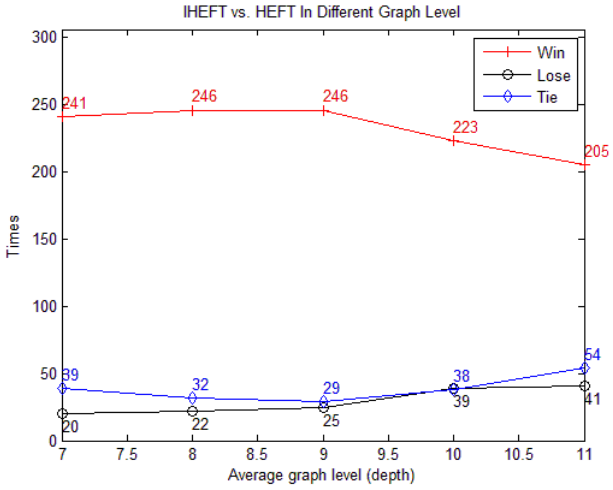

FIGURE 23. The times of win, lose or tie in different level

and average performance, they are still growing up when the specified level is reach in Fig. 25. We have done a further experiment for the high shape graphs about 20 levels, and the performance has fallen down to zero, it means that there is not any parallelism can be made, such that it can be better than HEFT.

Our final subtest focused on the various number of processor in a system. We first generate a number of DAG, and then fed to a different number of homogeneous processor in turn. In Fig. 26, we agree that two lines will start or stop at the same point, because there is minor different when the number of processor is small as well, or even no different when there is only one processor. On the other hand when the number of processor increase to a certain number, the critical path of DAG might dominate the execution time and it is not able to reduce any more. The cases of lose that keep growing up in this figure will break down the advantage of our algorithm or not, let us consider on the result depicting in Fig. 27. In that result, a conclusion we have drawn is that the performance approaches to zero, because our algorithm is modified from HEFT, it might have the same performance with HEFT or any other algorithm that cannot get over the critical path.

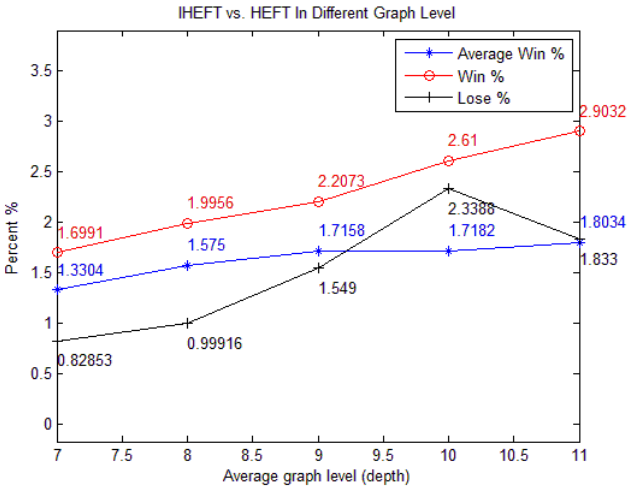

FIGURE 24. Time saving rate in different level

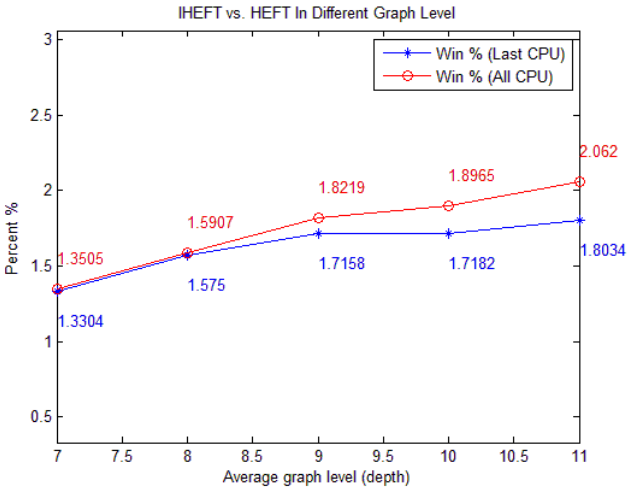

FIGURE 25. Total time saving in different level

However, we should take a look on the advantage that can let other processors be idle as soon as possible. Fig. 28 presents that the time saving from other sooner finished processors will bring us near 1.12% of total execution time.

6.4. **Experiment Result of IHEFT and Look-Ahead HEFT.** Here will present the result of performance evaluation between our algorithm and Look-ahead HEFT (LA-HEFT). The steps of this experiment are similar to the previous one, although the setting for this experiment might have a little bit difference. The properties of DAG are shown below:

The comparison procedures for this subtest are the same as what we have done before. We observe that LAHEFT is far better than our algorithm if we simply take account of the times of win, lose or tie in Fig. 29. However, if we observe Fig. 30, we find that the performance lines of both LAHEFT and IHEFT must almost twist together to form a horizontal and slightly wavy Average Win % line, and IHEFT behaved a little better than LAHEFT in general cases. Because Fig. 29 and Fig. 30 are from the same source, we can give a conclusion that although LAHEFT can really shorten the make-span to get over

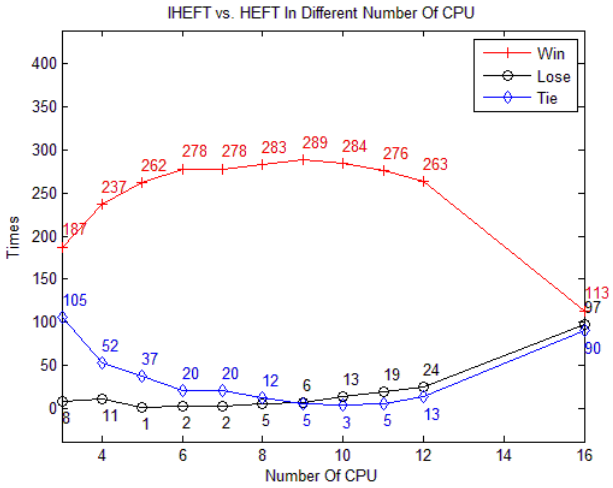

FIGURE 26. The times of each case in different number of CPU

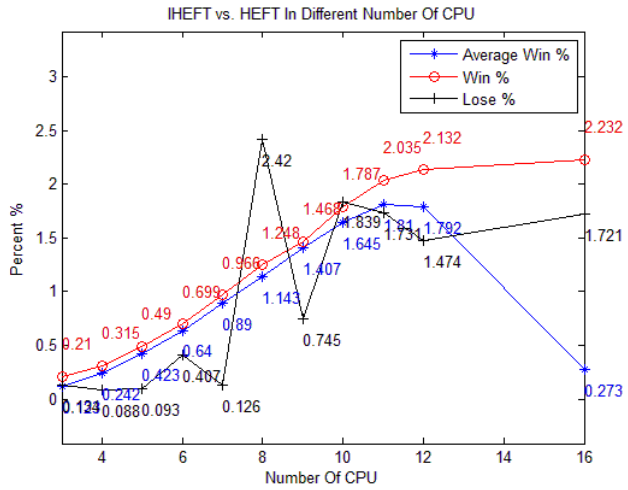

FIGURE 27. Time saving in different number of CPU

TABLE 4. Experiment setting of IHEFT - Look-ahead HEFT

Options	**Subtest 1**	**Subtest 2**	**Subtest 3**
Max. in degree	3	3	3
Min. level	3	**5 - 9**	10
Max. level	8	**9 - 13**	15
Min. number of node	**20 - 60**	70	400
Max. number of node	**40 - 80**	90	500
Min. node weight	10	100	10
Max. node weight	50	10000	50
Number of CPU	3	3	**3 - 20**

the other algorithms in general cases, but it cannot bring a very good accomplishment in final. Furthermore, in this subtest, let us take a look at Fig. 31, we even find that in the segment 50 - 60 nodes per graph, the total time saving percent of all CPU is positive,

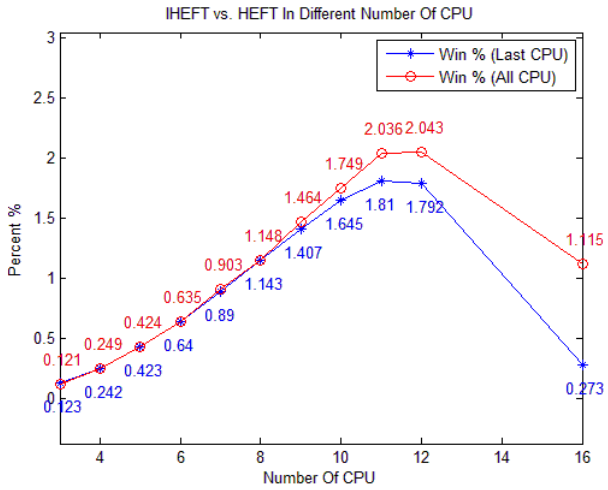

FIGURE 28. Total time saving in different number of CPU

although it is a negative value which only comparing the latest CPU. It means that if today we are focusing on the issues such as power consumption, processors idling overhead or parallel capability, we have not found any outstanding behavior are from LAHEFT.

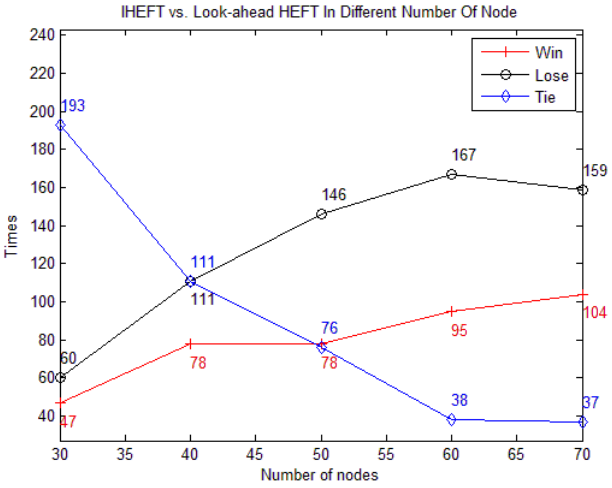

FIGURE 29. The times of win, lose or tie in difference number of nodes

The next subtest will take account of the affection of the changing of level. According to the times of win, lose and tie, we cannot recognize that what is the better, it just denote that what shape of the graphs might bring us the better result. We should be aware of the number of nodes in each level of this subtest is larger than the previous subtest, that might be the reason why the win ratio is larger than lose. However, the point which we are aiming at is the average winning percent at Fig. 33. It shows that the proposed algorithm is slightly better than LAHEFT. Actually, we do not care to save this so negligible time, but we will explain why our proposed algorithm is more reasonable choice in later section. Again, looking at the winning percent of all CPU at Fig. 34 can estimate the total average time that the proposed algorithm will save.

Finally, we have evaluated the change to performance of proposed algorithm against LAHEFT in different number of CPU. As shown in Fig. 35, according to the size of DAG,

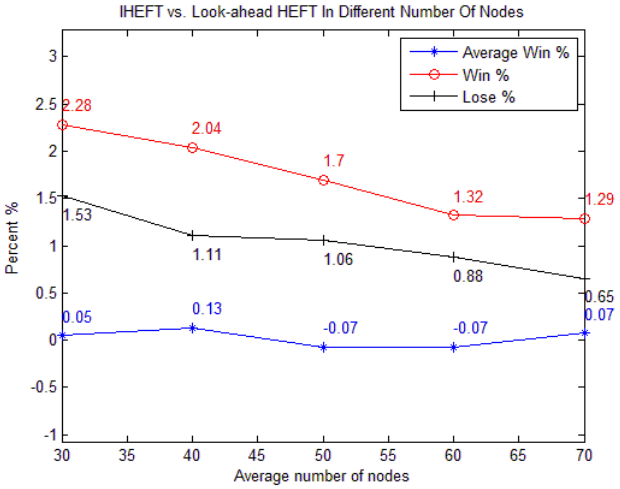

FIGURE 30. Time saving in different number of nodes

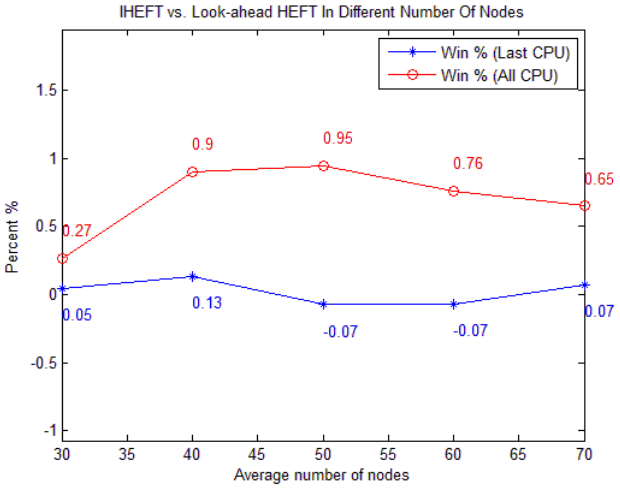

FIGURE 31. Total time saving of different number of nodes

when the CPU increase to a certain quantity, the times of win will be the largest, and it will fall down after that, and the times of tie keeps growing up later. This curve is what we can anticipate. For the average performance of our algorithm against LAHEFT, that might be the true colors of it, Fig. 36. Both algorithms start from zero (single processor), and LAHEFT performs a little bit better in a short moment after start and before end. However, the total time of saving by proposed algorithm is growing up dramatically in Fig. 37.

6.5. **Summary.** We have had a global view of our proposed algorithm after a series of comparing with HEFT and LAHEFT, and we found that it performs quite well in most of the cases. Because LAHEFT already performs very well in homogeneous systems, the ratio of speed up comparing with LAHEFT becomes not very obvious. On the other hand, in many different conditions that we have tested, the proposed algorithm always performs better than HEFT and almost approaches to optimal solutions when we observe the result at timing panel by naked eye.

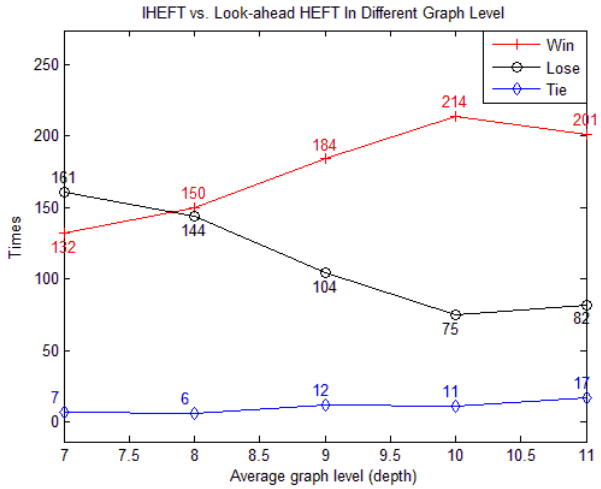

FIGURE 32. The times of win, lose or tie in different level

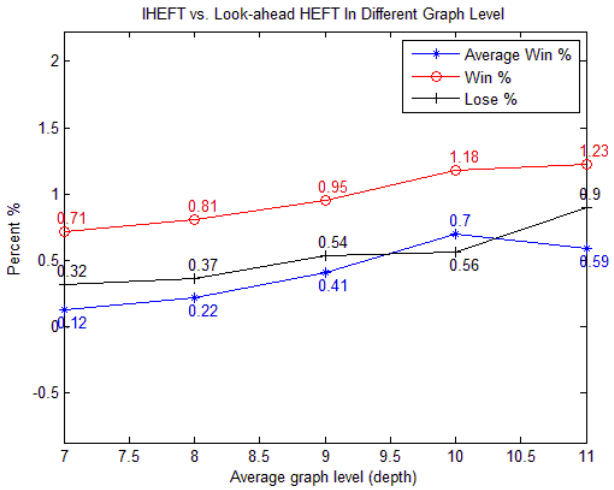

FIGURE 33. Time saving rate in different level

To boost up the performance for our algorithm, we have considered on involving the look-ahead technique to determine the best choice of processor (horizontally search) and the best inserted position (vertically search) for each node [19], but the algorithm procedure is very complex or even cannot be realized. The look-ahead technique cannot reduce the idle time during processing but search for the better solution by testing each processor, it might reduce the lose cases in our algorithm to minimize the make-span if we can combine these two ideas together.

When the tasks are required to be rescheduling after one node v_i has been inserted in front of node v_j in a queue, and the node v_j is possible to be reinserted again to the front of v_i, if this step takes place forever and ever, the system will halt. And hence, to avoid this ambiguous condition, we can set v_i as an immediate parent of v_j, such that v_j will never be rescheduled again in front of v_i, however, sometime it will make v_j cannot be rescheduled to the sooner time slot at another queue as well, so it performs worse in our experiments. By the way, the worst thing will happened if we try to recalculate the blevel and execution order of the nodes after assigning v_i as a parent of v_j, because that new edge might prolong the critical path of the graph accidentally. The recommended solution

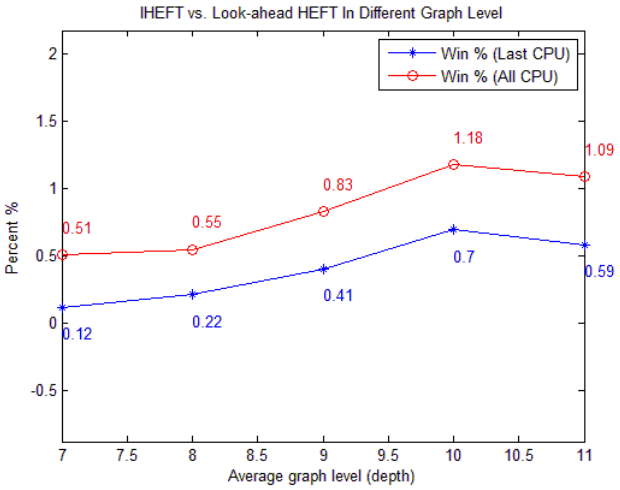

FIGURE 34. Total time saving in different level

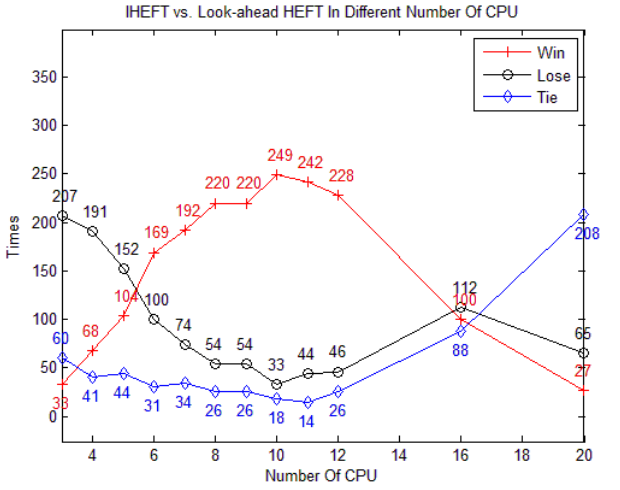

FIGURE 35. The times of each case in different number of CPU

is to keep using the first blevel and execution order for scheduling, and only rearrange the nodes if they really need. And finally, we have taken account of these constraints and brought out a solution for this problem, that is only trying to avoid v_j to be reinserted again before v_i by adding v_j to the list of sibling of v_i, because we assume that v_i found a better position after it had been inserted to the front of v_j.

7. **Conclusions.** In this paper, we have proposed a homogeneous multiprocessor system algorithm which is modified from HEFT. The main idea of this algorithm is trying to shorten the make-span by filling up all the timeslots of HEFT as much as possible, the supposed environment for running this algorithm is standalone homogeneous systems, and the communication cost is negligible or merge into the computation cost. On the other hand, we also discuss about the concept that how to parallel the vector graphics processing on a multiprocessor system, on this issue, we have provided the detail of the architecture of our system (single processor) as a sample for considering on the multiprocessor version.

Although our algorithm performs well in most cases comparing with HEFT and LA-HEFT, we still have to find out the better performance than look-ahead technique, and

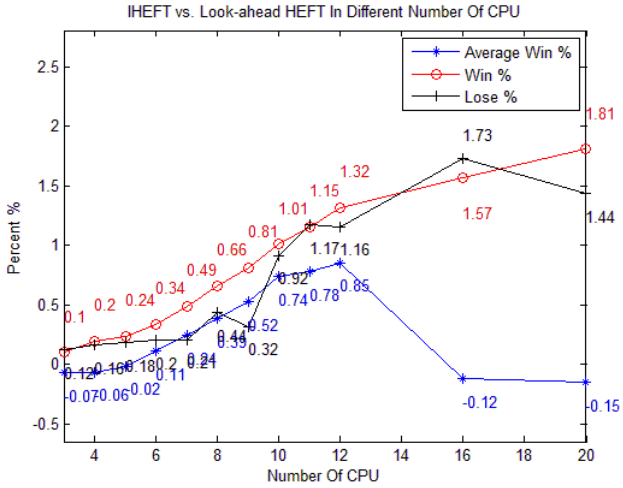

FIGURE 36. Time saving in different number of CPU

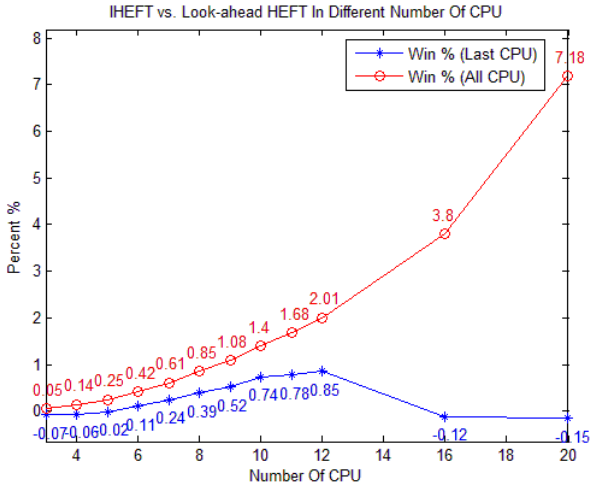

FIGURE 37. Total time saving in different number of CPU

also shorten the current execution time as well. We have not found a formal equation standing for the best performance of our algorithm if a DAG falls in this region, although we have compared a lot with HEFT and LAHEFT in this paper. In the most cases, our proposed algorithm behaves well for the homogeneous systems. We are wondering whether it is good for the heterogeneous systems as well, and what we should modify back to make it to be suitable to heterogeneous systems.

REFERENCES

[1] H. Topcuoglu, S. Hariri, and M. Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, 2002.

[2] H. Topcuoglu, S. Hariri, and M. Y. Wu, Task scheduling algorithms for heterogeneous processors, *Proc. of Eighth Heterogeneous Computing Workshop*, pp. 3-14, 1999.

[3] O. Sinnen, *Task Scheduling For Parallel Systems*, Canada, John Wiley & Sons, 2007.

[4] T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, 1994.

[5] I. Ahmad and Y. K. Kwok, On exploiting task duplication in parallel program scheduling, *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872-892, 1998.
[6] S. Ranaweera and D. P. Agrawal, A task duplication based scheduling algorithm for heterogeneous systems, *Proc. of Parallel and Distributed Processing Symposium*, pp. 445-450, 2000.
[7] X. Tang, K. Li, G. Liao, and R. Li, List scheduling with duplication for heterogeneous computing systems, *Journal of Parallel and Distributed Computing*, vol. 70, no. 4, pp. 323-329, 2010.
[8] G. Q. Liu, K. L. Poh, and M. Xie, Iterative list scheduling for heterogeneous computing, *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 654-665, 2005.
[9] T. L. Adam, K. M. Chandy, and J. R. Dickson, A comparison of list schedules for parallel processing systems *Communications of the ACM*, vol. 17, no. 12, pp. 685-690, 1974.
[10] Y. K. Kwok and I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381-422, 1999.
[11] M. A. Palis, J. C. Liou, and D. S. L. Wei, Task clustering and scheduling for distributed memory parallel architectures, *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46-55, 1996.
[12] R. C. Correa, A. Ferreira, and P. Rebreyend, Scheduling multiprocessor tasks with genetic algorithms, *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 8, pp. 825-837, 1999.
[13] A. S. Wu, H. Yu, S. Jin, K. C. Lin,and and G. Schiavone, An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling, *IEEE Trans. Parallel and Distributed Systems*, vol. 15, no. 9, pp. 824-834, 2004.
[14] L. Wang, H. J. Siegel, V. P. Roychowdhury, et al., Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8-22, 1997.
[15] O. Sinnen and L. Sousa, List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures, *Journal of Parallel Computing*, vol. 30, no. 1, pp. 81-101, 2004.
[16] D. Rice, Google Inc. , R. J. Simpson, AMD, et al., *OpenVG Specification Version 1.1*, The Khronos Group Inc., 2008.
[17] T. Hagras and J. Janecek, A High Performance, A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems, *Proc. of 18th International Parallel and Distributed Processing Symposium*, vol. 31, no. 7, pp. 653-670, 2005.
[18] J. Corbet, A. Rubini, and K. H. Greg, *Linux Device Driver third edition*, O'Reilly Media, 2005.
[19] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, *Proc. of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pp. 27-34, 2010.