A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention

Pengcheng Mu, Jean François Nezan, Mickael Raulet, Jean-Gabriel Cousin

A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention

Pengcheng Mu, Jean-François Nezan, Mickaël Raulet and Jean-Gabriel Cousin
IETR/Image and Remote Sensing Group
CNRS UMR 6164/INSA Rennes
20, avenue des Buttes de Coësmes
35043 RENNES Cedex, France
email: {pmu, jnezan, mraulet, jcousin}@insa-rennes.fr

Abstract

Task scheduling is an important aspect for parallel programming. In this paper, the program to be scheduled is modeled as a Directed Acyclic Graph (DAG), and we target parallel embedded systems of multiple processors connected by buses and switches. This paper presents improvements for list scheduling heuristics with communication contention. We use new node priorities ***(top level and bottom level)*** *to sort nodes and use an advanced technique of* ***critical child*** *to select a processor to execute a node. Experimental results show that our method is effective to reduce the schedule length, and the performance is greatly improved in the cases of medium and high communication. Since the communication cost is increasing from medium to high in modern applications like digital communication and video compression, our method will work well for scheduling these applications on parallel embedded systems.*

1. Introduction

The recent evolution of digital communication and video compression applications has dramatically increased the algorithm and system complexity. To face this problem, System on a Chip (SoC) with several cores (e.g. multi-core DSPs) and several hardware accelerators (e.g. Intellectual Properties) is becoming the basic element to build complex systems. It is not straightforward to distribute and schedule tasks of a program over a multi-component system. When performed manually, the result is usually a suboptimal solution. There is a need for new task scheduling methodologies that allow the exploration of several solutions over multiprocessor systems thus producing a near optimal result.

Dataflow programming has been commonly used for programming on multiprocessor. It consists in modeling a program as a directed graph of data flowing between operations. The program in this paper is represented as a Directed Acyclic Graph (DAG) [8], where nodes represent tasks (i.e. computations) and edges represent dataflows (i.e. communications) between tasks. The objective of task scheduling is to assign computations and communications respectively to processors and communication links of the target system in order to get the shortest execution time. The scheduling could be static, which is done at compile time, or dynamic, which is done at run time. Static scheduling is more suitable than dynamic scheduling for deterministic applications by leading to lower code size and higher computation efficiency. This paper concerns static scheduling, and all the task scheduling heuristics in the following parts are static.

The general task scheduling problem is proven to be NP-hard [8]; therefore, many works try to find heuristics to go up to the optimal solution. Early task scheduling heuristics as in [1, 4] do not consider communications. As the communication increases in modern applications, many scheduling heuristics have to take communication into account [8, 3, 14, 15, 6]. Most of these heuristics use fully connected topology network in which all communications can be performed concurrently. Different arbitrary processor networks are then used in [9, 5, 2, 12] to accurately describe real parallel systems, and the task scheduling takes into account communication contention on communication links.

Most of the heuristics above are based on the approach of list scheduling. Basic techniques are given in [10] for list scheduling with communication contention. This paper will give some advanced techniques. Firstly, three new groups of node priorities will be defined and used to sort nodes in addition to the two existing groups; secondly, a technique of using a node's ***critical child*** will be given to improve the performance for selecting a processor for this node. This paper will finally combine these two techniques and show

the efficiency in the results.

The paper is organized as follows: Section 2 firstly introduces necessary models and definitions, and then the task scheduling problem with communication contention is described in this section. Our new techniques are explained in Section 3 in detail, and Section 4 gives experimental results. The paper is concluded in Section 5 at last.

2. Models and Definitions

The program to be scheduled is called an algorithm and is modeled as a DAG in this paper. The multiprocessor target system is called an architecture and is modeled as a topology graph. These models are detailed as follows.

2.1. DAG Model

A DAG is a directed acyclic graph $G = (V, E, w, c)$ where V is the set of nodes and E is the set of edges. A node represents a computation. For two nodes $n_i, n_j \in V$, e_{ij} denotes the edge from the origin node n_i to the destination node n_j and represents the communication between these two computations. The weight of node n_i (denoted by $w(n_i)$) represents the computation cost; the weight of edge e_{ij}(denoted by $c(e_{ij})$) represents the communication cost. In this model, the set $\{n_x \in V : e_{xi} \in E\}$ of all the direct predecessors of node n_i is denoted by $pred(n_i)$; the set $\{n_x \in V : e_{ix} \in E\}$ of all the direct successors of node n_i is denoted by $succ(n_i)$. A node n with $pred(n) = \emptyset$ is named a source node, where $\emptyset$ is the empty set. A node n with $succ(n) = \emptyset$ is named a sink node.

The execution of computations on a processor is sequential and a computation can not be divided into several parts. A computation can not start until all its input communications finish, and all its output communications can not start until this computation finishes. Communications are also sequential on a communication link, but different computations and communications can be executed simultaneously respecting the input and output constraints above. Figure 1 gives a DAG example used in [7] to illustrate performances of different scheduling heuristics. It is also used in Section 4.1 to show the performance of our method.

2.2. Topology Graph Model

A topology graph $TG = (N, P, D, H, b)$ has been used to model a target system of multiple processors interconnected by communication links and switches [12]. N is the set of vertices, P is a subset of N, $P \subseteq N$, D is the set of directed edges, H is the set of hyperedges, and b is the relative data rate of edge. The union of the two edge sets D and H is designated the link set L, $L = D \cup H$, and an element of this set is denoted by l, $l \in L$.

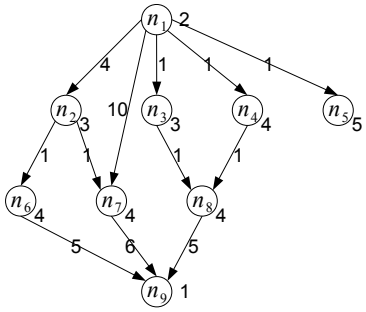


Figure 1. A DAG example

The topology graph is denoted as $TG = (N, P, L, b)$ in this paper, and directed edges are not used in a target system. A vertex $p \in P$ represents a processor, and a vertex $n \in N, n \notin P$ represents a switch. Since directed edges are not used, a link $l \in L$ is actually a hyperedge h, which is a subset of two or more vertices of N, $h \subseteq N, |h| > 1$. A hyperedge connects multiple vertices and represents a half duplex multidirectional communication link (e.g. a bus). The positive weight $b(l)$, associated with a link $l \in L$, represents its relative data rate.

Differing from the vertex of processor, a switch is a vertex used only for connecting communication links, and no computation can be executed on it. Switches are assumed to be ideal.

Ideal Switch For a switch s, let $l_1, l_2, \ldots, l_n$ be all the communication links connected to s. If two links l_{i_1} and l_{i_2} of them are not used for the moment, a communication can be transferred on l_{i_1} and l_{i_2} without any impact from/to communications on other communication links connected to s.

Switches are contention-free according to the description above. Separate communication links connected to the same switch can be used for different communications at the same time; however, a new communication could not begin on a link if this link is busy. Communication links are considered homogeneous in this paper, but processors can be heterogeneous. Therefore, the relative data rate is assumed to be 1 for all the links, $b(l) = 1, \forall l \in L$, but a computation usually needs different execution durations on different types of processors. Figure 2 gives three architecture examples: (a) three processors sharing a bus; (b) eight processors connected to a switch by eight buses; and (c) six processors interconnected by buses and switches. Figure 2(c) models the C6474 Evaluation Module (EVM)[1] which includes two C6474 multicore DSPs.

A route is used to transfer data from one processor to another in the target system. It is a chain of links connected

[1] http://focus.ti.com/docs/toolsw/folders/print/tmdxevm6474.html

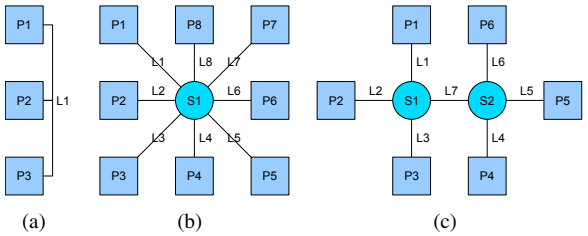


Figure 2. Architecture examples

by switches from the origin processor to the destination processor. For example, $L1 \rightarrow L7 \rightarrow L4$ is a route from $P1$ to $P4$ in Figure 2(c). Routing is an important aspect of task scheduling. Since the scheduling is static, a route between two processors is also considered as static and is determined at compile time. It is possible to determine routes once and to store them in a table, then the routing during the scheduling becomes looking up the table.

2.3. Task Scheduling with Communication Contention

A schedule of a DAG is the association of a start time and a processor with each node of the DAG. When the communication contention is considered, a schedule also includes allocating communications to links and associating start times on these links with each communication. A communication needs the same duration on each link because of the homogeneity of links. However, a computation usually needs different durations on different processors because processors are heterogeneous. Therefore, the average duration of a computation on different types of processors is used to present the node weight.

Following terms describe a schedule S of a DAG $G = (V, E, w, c)$ over a topology graph $TG = (N, P, L, b)$. The start time of a node $n_i \in V$ on a processor $p \in P$ is denoted by $t_s(n_i, p)$; the finish time is given by $t_f(n_i, p) = t_s(n_i, p) + w(n_i, p)$, where $w(n_i, p)$ is the execution duration of n_i on p. A node can be constrained to some processors of the target system. The set of processors on which n_i can be executed is denoted by $Proc(n_i)$, and the processor on which n_i is actually allocated is denoted by $proc(n_i)$. The finish time of a processor is the maximum finish time among all the nodes allocated on this processor, $t_f(p) = \max_{proc(n_i)=p} \{t_f(n_i, proc(n_i))\}$, and the schedule length of S is the maximum finish time among all the processors in the system, $sl(S) = \max_{p \in P} \{t_f(p)\}$.

The communication represented by an edge exists only when its origin node and destination node are not allocated on the same processor. The start time of an existing edge $e_{ij} \in E$ on a link $l \in L$ is denoted by $t_s(e_{ij}, l)$; the finish time of e_{ij} is given by $t_f(e_{ij}, l) = t_s(e_{ij}, l) + c(e_{ij})$.

A node (computation) can start on a processor at the time when all the node's input edges (communications) finish. This time is called the Data Ready Time (DRT) and is denoted by $t_{dr}(n_j, p) = \max_{e_{ij} \in E} \{t_f(e_{ij}, l)\}$, where l is a link on which e_{ij} is allocated. DRT is the earliest time when a node can start. If n_j is a node without input edge, $t_{dr}(n_j, p) = 0, \forall p \in P$.

Node Scheduling Condition For a node n_i, let $[A, B], A, B \in [0, \infty]$ be an idle time interval on the processor p. n_i can be scheduled on p within $[A, B]$ if $\max\{A, t_{dr}(n_i, p)\} + w(n_i, p) \leq B$. The start time of n_i on p is given by $t_s(n_i, p) = \max\{A, t_{dr}(n_i, p)\}$.

Communications are handled in the way of cut-through on a route because of the circuit switching. Therefore, an edge e_{ij} is aligned on all the links of the route $l_{R_1} \rightarrow l_{R_2} \rightarrow \ldots \rightarrow l_{R_k}$ with $t_s(e_{ij}, l_{R_1}) = t_s(e_{ij}, l_{R_2}) = \ldots = t_s(e_{ij}, l_{R_k})$. The start time and finish time of e_{ij} on all the links of the route are denoted uniformly by $t_s(e_{ij})$ and $t_f(e_{ij})$ with $t_f(e_{ij}) = t_s(e_{ij}) + c(e_{ij})$.

Edge Scheduling Condition For a DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$, let $l_{R_1} \rightarrow l_{R_2} \rightarrow \ldots \rightarrow l_{R_k}$ be a route for an edge $e_{ij} \in E$ and let $[A, B], A, B \in [0, \infty]$ be a common idle time interval on all the links of this route. e_{ij} can be scheduled on this route within $[A, B]$ if $\max\{A, t_f(n_i, proc(n_i))\} + c(e_{ij}) \leq B$. The start time of e_{ij} on this route is given by $t_s(e_{ij}) = \max\{A, t_f(n_i, proc(n_i))\}$.

3. List Scheduling Heuristic

Algorithm 1 gives the commonly used static list scheduling heuristic. This algorithm is composed of three procedures of `Sort_Nodes()`, `Select_Processor()` and `Schedule_Node()`. This section describes improvements for the first two procedures compared with the classic methods given in [12].

Algorithm 1: **List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

```
1 NodeList ← Sort_Nodes(V);
2 for each n ∈ NodeList do
3 |   p_best ← Select_Processor(n, P);
4 |   Schedule_Node(n, p_best);
5 end
```

3.1. Sorting Nodes with Five Groups of Node Priorities

Nodes are firstly sorted into a static list by the procedure of `Sort_Nodes()` in the heuristic. Since the order of nodes in the list affects much the schedule result, many different priority schemes have been proposed to sort nodes [9, 6]. Experiments in [11] show that list scheduling with static list sorted by bottom level outperforms other compared contention aware algorithms. Our list scheduling heuristic uses the bottom level and top level to sort nodes, and three new groups of top level and bottom level are proposed to take communication contention into account.

The top level of a node is the length of the longest path from the beginning of the DAG to this node, excluding the weight of this node; the bottom level of a node is the length of the longest path from this node to the end of the DAG, including the weight of this node. Our procedure of `Sort_Nodes()` sorts nodes into a list of $NodeList$ according to the following rule:

Rule for Sorting Nodes Nodes are sorted by the decreasing order of their bottom levels; if two nodes have equal bottom levels, the one with greater top level is placed before the other; if both the bottom level and the top level are equal, these nodes are sorted randomly.

Two groups of top level and bottom level have been used as node priorities and are named respectively as computation top level (tl_{comp}) and bottom level (bl_{comp}), top level (tl) and bottom level(bl). Besides the two existing groups, this paper proposes three new groups, which are named as input top level (tl_{in}) and bottom level (bl_{in}), output top level (tl_{out}) and bottom level (bl_{out}), input/output top level (tl_{io}) and bottom level (bl_{io}). Figure 3 illustrates the dependency between nodes to define different top levels and bottom levels, where the dotted nodes and edges are used to define the top levels and bottom levels of n_i. The formalized definitions of top levels and bottom levels are given as follows.

- Computation top level and bottom level (Figure 3(a))

  The computation top level of a node is the length of the longest path from the beginning of the DAG to this node including only the weights of nodes; the computation bottom level of a node is the length of the longest path from this node to the end of the DAG including only the weights of nodes. The weights of edges are not taken into account in the computation top level and bottom level. They are defined recursively as follows:

$$tl_{comp}(n_i) = \begin{cases} 0, \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl_{comp}(n_k) + w(n_k)\}, \\ \text{others} \end{cases}$$

$$bl_{comp}(n_i) = \begin{cases} w(n_i), \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl_{comp}(n_k)\} + w(n_i), \\ \text{others} \end{cases}$$

- Top level and bottom level (Figure 3(b))

  The top level and bottom level take into account additionally the weights of edges on the path by contrast with the computation top level and bottom level. They are defined recursively as follows:

$$tl(n_i) = \begin{cases} 0, \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl(n_k) + w(n_k) + c(e_{ki})\}, \\ \text{others} \end{cases}$$

$$bl(n_i) = \begin{cases} w(n_i), \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl(n_k) + c(e_{ik})\} + w(n_i), \\ \text{others} \end{cases}$$

- Input top level and bottom level (Figure 3(c))

  The input top level and bottom level take into account weights of nodes on the path as well as weights of all the input edges of a node on the path. They are defined recursively in Equation 1 and 2.

- Output top level and bottom level (Figure 3(d))

  The output top level and bottom level take into account weights of nodes on the path as well as weights of all the output edges of a node on the path. They are defined recursively in Equation 3 and 4.

- Input/output top level and bottom level (Figure 3(e))

  The input/output top level and bottom level take into account weights of nodes on the path as well as weights of all the input and output edges of a node on the path. They are defined recursively in Equation 5 and 6.

The three new priorities take into account the communication contention between nodes in comparison with the two existing priorities which have been used in the list scheduling without communication contention. Table 1 gives all the five groups of top levels, bottom levels and the resulting static lists for the DAG given in Figure 1. Since the bottom level reflects the time needed from this node to the end of the graph, our new bottom levels reflect better the reality in the case of communication contention. Experiments in Section 4 will show that using the combination of these priorities improves the performance for list scheduling with communication contention.

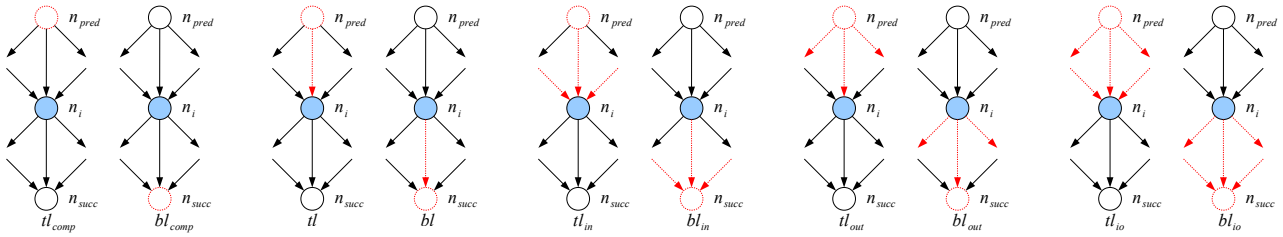


Figure 3. Five groups of node priorities

$$tl_{in}(n_i) = \begin{cases} 0, \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl_{in}(n_k) + w(n_k)\} + \sum\limits_{e_{li} \in E} c(e_{li}), \text{others} \end{cases} \quad (1)$$

$$bl_{in}(n_i) = \begin{cases} w(n_i), \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{in}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) \right\} + w(n_i), \text{others} \end{cases} \quad (2)$$

$$tl_{out}(n_i) = \begin{cases} 0, \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{out}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) \right\}, \text{others} \end{cases} \quad (3)$$

$$bl_{out}(n_i) = \begin{cases} w(n_i), \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl_{out}(n_k)\} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), \text{others} \end{cases} \quad (4)$$

$$tl_{io}(n_i) = \begin{cases} 0, \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{io}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) - c(e_{ki}) \right\} + \sum\limits_{e_{li} \in E} c(e_{li}), \text{others} \end{cases} \quad (5)$$

$$bl_{io}(n_i) = \begin{cases} w(n_i), \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{io}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) - c(e_{ki}) \right\} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), \text{others} \end{cases} \quad (6)$$

3.2. Processor Selection

Classic list scheduling heuristics select the processor allowing the earliest finish time for a node. This rule gives probably a locally optimized result. The critical child of a node is used to solve this problem for scheduling with unbounded number of processors in [6]. Our paper uses the concept of critical child for list scheduling with bounded number of processors in the case of communication contention. The critical child is defined differently as follows:

Critical Child Given a static node list $NodeList$, the critical child of node n_i is denoted by $cc(n_i)$ and is one of n_i's successors which emerges firstly in $NodeList$.

The critical child of n_i may be different if $NodeList$ differs. Using critical child makes the processor selection take into account not only the predecessors of a node, but also its most important successor. Our method of using the critical child to select processor is given in Algorithm 2. Since it is possible that $cc(n_i)$ is not a free node with all its predecessors scheduled during the processor selection for n_i, the scheduling of $cc(n_i)$ only takes into account its scheduled predecessors in the procedure of `Select_Processor()` for n_i.

3.3. Node and Edge Scheduling

The method of scheduling a node n_i onto a processor p is given in Algorithm 3, and Algorithm 4 gives the method for edge scheduling. Since an edge e_{ij} is scheduled only when its origin node n_i has been scheduled, the scheduling of this edge needs additionally the processor p on which the destination node n_j of e_{ij} to be scheduled.

Table 1. Different levels and static orders

n_i	tl_{comp}	bl_{comp}	order	tl	bl	order	tl_{in}	bl_{in}	order	tl_{out}	bl_{out}	order	tl_{io}	bl_{io}	order
n_1	0	11	(1)	0	23	(1)	0	41	(1)	0	35	(1)	0	55	(1)
n_2	2	8	(4)	6	15	(2)	6	35	(2)	19	16	(2)	19	36	(2)
n_3	2	8	(3)	3	14	(4)	3	26	(4)	19	14	(4)	19	26	(4)
n_4	2	9	(2)	3	15	(3)	3	27	(3)	19	15	(3)	19	27	(3)
n_5	2	5	(8)	3	5	(8)	3	5	(8)	19	5	(8)	19	5	(8)
n_6	5	5	(7)	10	10	(6)	11	21	(6)	24	10	(7)	24	21	(7)
n_7	5	5	(6)	12	11	(5)	20	21	(5)	24	11	(5)	34	21	(5)
n_8	6	5	(5)	8	10	(7)	9	21	(7)	24	10	(6)	25	21	(6)
n_9	10	1	(9)	22	1	(9)	40	1	(9)	34	1	(9)	54	1	(9)
	NodeList_1			**NodeList_2**			**NodeList_2**			**NodeList_3**			**NodeList_3**		

Algorithm 2: **Select_Processor**(n_i, P)

Input: A node $n_i \in V$ and the set P of all processors
Output: The best processor p_{best} for the input node n_i

Choose the critical child $cc(n_i)$;
$BestFinishTime \leftarrow \infty$;
for *each* $p \in Proc(n_i)$ **do**
  $FinishTime \leftarrow$ **Schedule_Node**(n_i, p);
  $MinFinishTime \leftarrow \infty$;
  if $cc(n_i) \neq null$ **then**
    for *each* $p' \in Proc(cc(n_i))$ **do**
      $FinishTime \leftarrow$ **Schedule_Node**($cc(n_i)$, p');
      if $FinishTime < MinFinishTime$ **then**
        $MinFinishTime \leftarrow FinishTime$;
      end
    end
  else
    $MinFinishTime \leftarrow FinishTime$;
  end
  if $MinFinishTime < BestFinishTime$ **then**
    $BestFinishTime \leftarrow MinFinishTime$;
    $p_{best} \leftarrow p$;
  end
end

Algorithm 3: **Schedule_Node**(n_i, p)

Input: $n_i \in V$ and a processor $p \in P$
Output: The finish time of n_i on p

for *each* $n_l \in pred(n_i), proc(n_l) \neq p$ **do**
  Schedule_Edge(e_{li}, p);
end
Calculate DRT of node n_i;
Find the earliest idle time interval for node n_i on processor p respecting the node scheduling condition;
Calculate the finish time of n_i on p;

Algorithm 4: **Schedule_Edge**(e_{ij}, p)

Input: $e_{ij} \in E$ and a processor $p \in P$ on which the node n_j is to be scheduled
Output: None

if n_i *is scheduled* **then**
  if $proc(n_i) \neq p$ **then**
    Determine the route R from $proc(n_i)$ to p;
    Find the earliest common idle time interval on all the links of R respecting the edge scheduling condition;
  end
end

4. Experimental Results

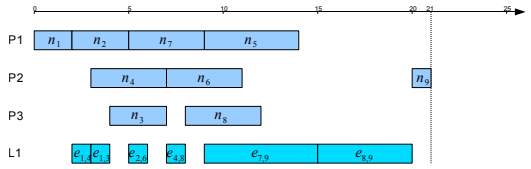
This section gives experimental results of our proposed list scheduling heuristic compared to the classic one given in [12]. The architecture in Figure 2(a) and 2(b) are respectively used for the comparison in Section 4.1 and 4.2.

4.1. Comparison with an Example

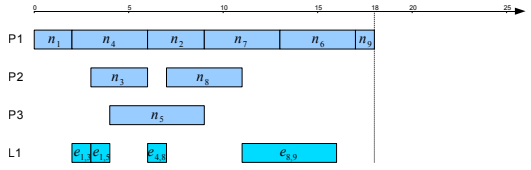
The DAG given in Figure 1 is used in this section to show that using the critical child and different priorities improves the schedule performance. Table 1 has given all the five groups of top levels, bottom levels and the resulting static lists for this DAG. The critical child for each node is obtained according to these static lists.

Figure 4(a) gives the schedule result of the classic heuristic with nodes sorted by bl and tl, and the schedule length is 21. Using the critical child technique with the three different node lists in Table 1 gives different schedule results. The schedule result for the node list sorted by bl_{comp} and tl_{comp} is shown in Figure 4(b) with the schedule length of 18. Since the node list sorted by bl and tl is same as that sorted by bl_{in} and tl_{in}, the same schedule result is obtained

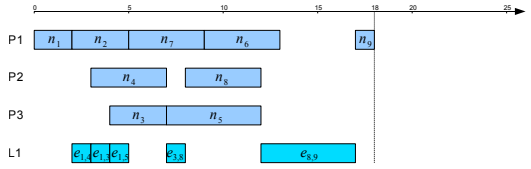
and shown in Figure 4(c) with the schedule length of 18. Figure 4(d) gives the schedule result for the same node list sorted by bl_{out} and tl_{out} and by bl_{io} and tl_{io}. The schedule length is 17 and is better than the two former schedule lengths of 18. All the three schedule results of using the critical child technique are better than that of the classic heuristic.



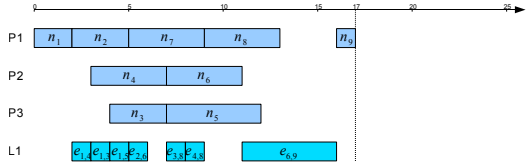
(a) Classic heuristic



(b) Critical child with NodeList_1



(c) Critical child with NodeList_2



(d) Critical child with NodeList_3

Figure 4. Schedule results

4.2. Comparison with Random DAGs

Random graphs are commonly used to compare scheduling algorithms in order to get statistical results which are more persuasive than the result for a particular graph. We implement a graph generator based on SDF^3 [13] to generate random SDF graphs except that the SDF graphs are constrained to be DAGs (same rate between two operations, no cycles). A random DAG is described in five aspects: the number of nodes, the average in degree, the average out degree, the random weights of nodes and the random weights of edges. The average in degree and out degree are assumed to be same. The weights of nodes vary randomly from w_{min} to w_{max}. The communication to computation ratio (CCR) is used to generate random weights of edges. The CCR is defined as the average weight of edges divided by the average weight of nodes in this paper, that is $CCR = \frac{\frac{1}{|E|}\sum_{e \in E} c(e)}{\frac{1}{|V|}\sum_{n \in V} w(n)}$. The weights of edges are generated randomly from $w_{min} \times CCR$ to $w_{max} \times CCR$. The CCR's typical values of 0.1, 1 and 10 represent respectively the low, medium and high communication situations.

A list scheduling heuristic can use all the five groups of node priorities to get different results. We combine the five groups of node priorities with a heuristic and choose the best result; the whole process is called a combined heuristic. The schedule length of the combined heuristic is compared to the classic list scheduling heuristic with nodes sorted by bl and tl. The acceleration factor (acc) is defined as $acc = \frac{sl_{classic}}{sl_{compared}}$ to show the speed-up of the compared heuristic.

Figure 5 gives the average acc of the combined heuristic with critical child. Weights of nodes are generated randomly from 100 to 1000, and 1000 random DAGs for each group are tested to obtain the statistical results.

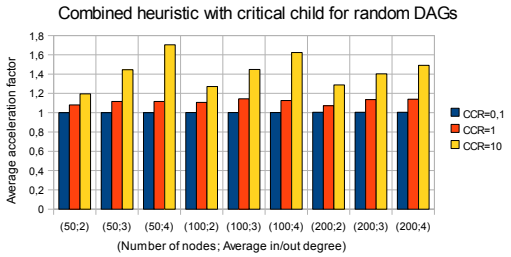


Figure 5. Average acc of combined heuristic with critical child

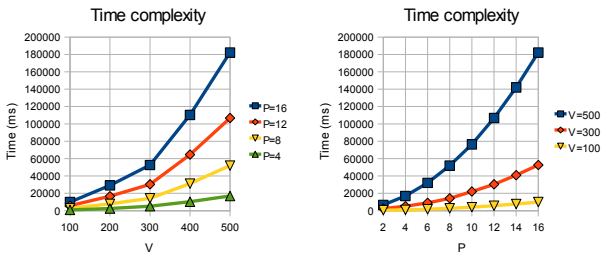
The average acc increases as CCR increases, and the schedule result is sped up by using the combined heuristic in the cases of $CCR = 1$ and $CCR = 10$. The average acc also increases as the number of average in/out degree increases when $CCR = 10$. The reason for this phenomenon is that the critical child technique helps to select better processors for nodes with multiple predecessors. The greater the in/out degree is, the better the critical child works. Since the modern applications like digital communication and video compression usually have $CCR > 1$, our method will be suitable for scheduling these applications on parallel embedded systems.

4.3. Time Complexity

The classic list scheduling heuristic has the time complexity of $O\left(PE^2O\left(routing\right) + V^2\right)$, where P, V and E are respectively the number of processors, the number of nodes and the number of edges. The time complexity increases by a factor of P by using the critical child, but the

combination with different node priorities does not increase the time complexity. Therefore, the time complexity of our combined heuristic is $O\left(P\left(PE^2O\left(routing\right)+V^2\right)\right)$.

Figure 6 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined heuristic. All the DAGs have the average in/out degree of 4, and all the processors are connected to a switch. As shown in Figure 6(a) and Figure 6(b), the time increases as the square of V and also as the square of P. We run our heuristic on a Pentium Dual-Core PC at 2.4GHz, and it takes about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors. In fact, a complicated embedded application usually has less than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Therefore, the increase of time complexity is reasonable and acceptable for rapid prototyping.



(a) (b)

Figure 6. Time complexity

5. Conclusions

This paper presents three new groups of node priorities (top level and bottom level) and a technique of critical child for list scheduling with communication contention. The new priorities take the communication contention into account and are used to sort nodes in order to get different node lists. The technique of critical child helps to select a better processor for a node. The combination of different node lists and the critical child technique gives different schedule results for a given DAG, and the best one is chosen at last. Experimental results show that using different node lists and the critical child technique is effective to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication. Since the communication cost is increasing from medium to high in modern digital communication and video compression applications, our method will work well for scheduling these applications on embedded parallel systems.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[2] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[3] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[4] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comput.*, 33(11):1023–1029, 1984.

[5] Y.-K. Kwok and I. Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 36, Washington, DC, USA, 1995. IEEE Computer Society.

[6] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs onto multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[7] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[8] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, 1989.

[9] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4:175–187, Feb. 1993.

[10] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007.

[11] O. Sinnen and L. Sousa. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, Jan. 2004.

[12] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, June 2005.

[13] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[14] M.-Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[15] T. Yang and A. Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, Sept. 1994.