Static vs. Dynamic List-Scheduling Performance Comparison

T. Hagras, J. Janeček

The problem of efficient task scheduling is one of the most important and most difficult issues in homogeneous computing environments. Finding an optimal solution for a scheduling problem is NP-complete. Therefore, it is necessary to have heuristics to find a reasonably good schedule rather than evaluate all possible schedules. List-scheduling is generally accepted as an attractive approach, since it pairs low complexity with good results. List-scheduling algorithms schedule tasks in order of priority. This priority can be computed either statically (before scheduling) or dynamically (during scheduling). This paper presents the characteristics of the two main static and the two main dynamic list-scheduling algorithms. It also compares their performance in dealing with random generated graphs with various characteristics.

Keywords: list scheduling, compile time scheduling, task graph scheduling, homogeneous computing.

1 Introduction

Efficient scheduling of computationally intensive programs is one of the most essential and most difficult issues to achieve high performance in a homogeneous computing environment [1]. When the characteristics of an application are known a priori, including tasks execution time, data size of communication between tasks, and task dependencies, the application is represented by a static model [2]. In the static model, the application is represented by a directed acyclic graph (DAG) in which the nodes represent the application tasks and the edges represent intertask data dependencies. Each node is labeled by the computation cost (expected computation time) of the task and each edge is labeled by the communication cost (expected communication time) between tasks [3, 4, 5, 6, 7].

The objective of scheduling is to map the tasks onto the processors (machines) and order their execution so the that task dependencies are satisfied and minimum overall scheduling length (makespan) is achieved. Finding an optimal solution for the scheduling problem is NP-complete [3, 4, 5, 6, 7]. Therefore, it is necessary to have heuristics to find the best scheduling rather than evaluate all possible scheduling combinations.

Most scheduling heuristics algorithms are based on list-scheduling [2, 3, 4, 7]. List-scheduling consists of two phases: *a task prioritizing phase*, where a priority is computed and assigned to each node of the DAG, and *a processor selection phase*, where each task (in order of its priority) is assigned a processor that minimizes a suitable cost function. The scheduling heuristic is called *static* if the processor selection phase starts after completion of the task prioritizing phase [2, 8] and it is called *dynamic* if the two phases are interleaved [9, 10].

This paper presents the characteristics of the two main static and the two main dynamic list-scheduling algorithms. It also compares their performance over a 90K variant random graph. The remainder of this paper is organized as follows. The next section defines the static task-scheduling problem and gives the background of the problem including some definitions and parameters used in the algorithms. Section 3 presents a brief review of the examined algorithms. Section 4, presents a performance comparison of the reviewed algorithm. Section 5 provides the conclusion.

2 Task Scheduling Problem

This section presents the application model used for static task scheduling and the homogeneous computing environments model that they will be used for the surveyed algorithms. The application can be represented by a *directed acyclic graph* $G(V, E, C, W)$ where:

V is the set of v nodes, and each node $v_i \in V$ represents an application task, which is a sequence of instructions that must be executed serially on the same processor,

W is the set of computation costs, where $w_i \in W$ is the execution time of task v_i,

E is the set of communication edges. The directed edge $e_{i,j}$ joins nodes v_i and v_j, where node v_i is called the parent node and node v_j is called the child node. This also implies that v_j cannot start until v_i finishes and sends its data to v_j.

C is the set of communication costs, and the edge $e_{i,j}$ has a communication cost $c_{i,j} \in C$.

A task without any parent is called an *entry task* and a task without any child is called an *exit task*. If there is more than one exit (entry) task, they may be connected to a zero-cost pseudo exit (entry) task with zero-cost edges, which do not affect the schedule.

The homogeneous computing environment model is a set P of p identical processors connected in a fully connected graph. It is also assumed that:

- any processor can execute the task and communicate with other processors at the same time,
- once a processor has started task execution, it continues without interruption, and on completing the execution it sends immediately the output data to all children tasks in parallel.

The communication cost $c_{i,j}$ for transferring data from task v_i (scheduled on p_m) to task v_j (scheduled on p_n), is defined as:

$$c_{i,j} = S + R \cdot \mu_{i,j},$$

where

S	is the cost of starting communication between processors (in secs),
$\mu_{i,j}$	is the amount of data transmitted from task v_i to task v_j (in bytes),
R	is the cost of communication per transferred byte (in sec/byte).

It is assumed that startup cost S is negligible and the unit cost R is the same for any two processors, so that the communication cost for any two tasks is a function of the amount of transferred data only.

2.1 Basic Scheduling Attributes

The frequently used attributes for assigning priority in list-scheduling are $t-level$ (top level) and $b-level$ (bottom level). The $t-level$ of node v_i is the length of the longest path from the entry node to v_i (excluding v_i). Here, the length of the path is the sum of all nodes and edges weights along the path. The $t-level(v_i)$ is computed recursively by traversing the DAG downward starting from the entry node v_{entry} as follows:

$$t-level(v_i) = \max_{v_m \in pred(v_i)} \left\{ t-level(v_m) + w_m + c_{m,i} \right\},$$

where $pred(v_i)$ is the set of immediate predecessors of v_i and $t-level(v_{entry}) = 0$.

The $b-level(v_i)$ of node v_i is the length of the longest path from v_i to the exit node. The $b-level(v_i)$ is computed recursively by traversing the DAG upward starting from the exit node v_{exit} as follows:

$$b-level(v_i) = w_i + \max_{v_m \in succ(v_i)} \left\{ b-level(v_m) + c_{i,m} \right\},$$

where $succ(v_i)$ is the set of immediate successors of v_i and $b-level(v_{exit}) = w(v_{exit})$.

If the edge weights are not taken into account in computing the $b-level$, the $b-level$ in this case is called the *static* $b-level$ or simply the *static level* (*SL*). The *SL* can be computed recursively by traversing the DAG upward starting from the exit node v_{exit} as follows:

$$SL(v_i) = w_i + \max_{v_m \in succ(v_i)} \left\{ SL(v_m) \right\},$$

where $succ(v_i)$ is the set of immediate successors of v_i and $SL(v_{exit}) = w(v_{exit})$.

Two other attributes are also used to assign priority to the nodes: *EST* (Earliest Start Time), also called *ASAP* (As Soon As Possible), and *LST* (Latest Start Time), also called *ALAP* (As Late As Possible).

The $EST(v_i)$ of v_i is highly correlated with the $t-level$ of v_i and the procedure for computing the $t-level$ can be used to compute the nodes earliest start times. The *EST* can be computed recursively by traversing the DAG downward starting from the entry node v_{exit} as follows:

$$EST(v_i) = \max_{v_m \in pred(v_i)} \left\{ EST(v_m) + w_m + c_{m,i} \right\},$$

where $pred(v_i)$ is the set of immediate predecessors of v_i and $EST(v_{entry}) = 0$.

The *LST* can be computed recursively by traversing the DAG upward, starting from the exit node v_{exit}, as follows:

$$LST(v_i) = \min_{v_m \in succ(v_i)} \left\{ LST(v_m) - c_{i,m} \right\} - w_i,$$

where $succ(v_i)$ is the set of immediate successors of v_i and $LST(v_{exit}) = EST(v_{exit})$.

The critical path (CP) of a DAG is the longest path from the entry node to the exit node. Clearly a DAG can have more than one CP. Consider the DAG shown in Fig. 1, where each node has two labels, the upper one is indicating the node label and the lower one is indicating the node weight. In this DAG, the nodes v_1, v_2, v_9, v_{10} are the CP nodes and are called CPNs (Critical Path Nodes). The edges on the CP are shown by thick arrows. The values of the priorities discussed above are shown in Table 1.

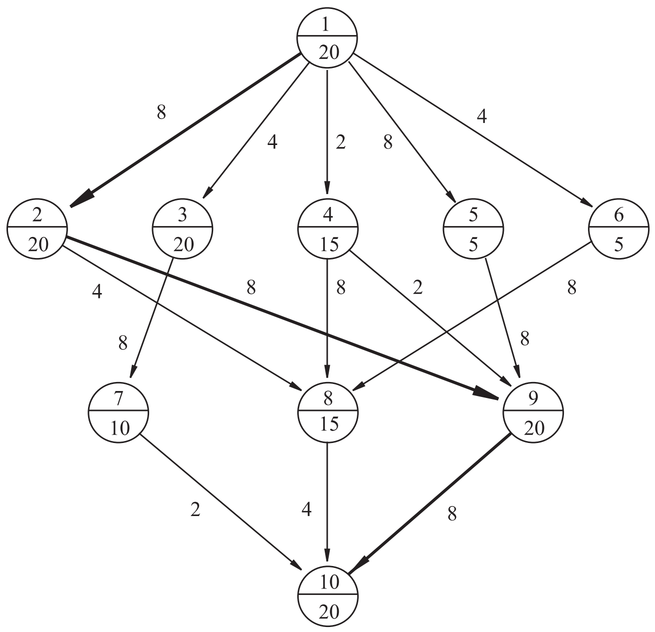

Fig. 1: Application directed acyclic graph (DAG)

Table 1: Priority attributes

Node	*SL*	$t-level$ & *EST*	$b-level$	*LST*
1	80	0	104	0
2	60	28	76	28
3	50	24	60	44
4	55	22	65	39
5	45	28	61	43
6	40	24	52	52
7	30	52	32	72
8	35	52	39	65
9	40	56	48	56
10	20	84	20	84

3 List-Scheduling Algorithms

This section presents the two main static list-scheduling algorithms and the two main dynamic list-scheduling algorithms. All these algorithms are for a limited number of homogeneous processors.

3.1 Static List-Scheduling Algorithms

This section briefly reviews the two main static list-scheduling algorithms, which are the Highest Level First with Estimated Time (HLFET) algorithm [2] and the Modified Critical Path (MCP) algorithm [8].

3.1.1 HLFET algorithm

The Highest Level First with Estimated Time (HLFET) algorithms [2] is one of the simplest list-scheduling algorithms and is described as follows in Fig. 2.

```
1. Compute the SL (static level) for each node in the graph
2. Put all nodes in a list L and sort L in a descending order of nodes SL
3. while not the end L do
       – dequeue $v_i$ from L
       – compute earliest execution start time for $v_i$ in all processors
       – schedule $v_i$ to the processor that minimizes the node earliest execution start time
```

Fig. 2: HLFET algorithm

The complexity of the HLFET algorithm is $O(pv^2)$. For the DAG shown in Fig. 1, the scheduling trace of HLFET algorithm is given in Table 2. In the table, the execution start times of each node on all available processors at each step are given, and the nodes on the list are scheduled one by one, to the processor that allows the earliest execution start time.

Table 2: A scheduling trace of the HLFET algorithm (makespan = 88)

Step	*Selected v*	p_1	p_2	p_3	*Selected p*
1	1	0	0	0	p_1
2	2	20	28	28	p_1
3	4	40	22	22	p_2
4	3	40	37	24	p_3
5	5	40	37	44	p_2
6	6	40	42	44	p_1
7	9	50	48	50	p_2
8	8	45	68	53	p_1
9	7	60	68	44	p_3
10	10	76	68	76	p_2

3.1.2 MCP Algorithm

The Modified Critical Path (MCP) algorithm [8] uses the *ALAP* attribute (*LST* defined in section 2) of a node as the scheduling priority. The MCP algorithm first computes the *ALAPs* of all nodes, and then constructs a list of nodes in ascending order of nodes *ALAP*. In the case of equivalent *ALAP* values, the *ALAPs* of the children are taken into consideration to break the tie. The MCP algorithm then schedules the nodes on the list one by one such that a node is scheduled to the processor that allows the earliest execution start time. The MCP algorithm is shown in Fig. 3.

```
1. Compute the ALAP (LST in section 2) for each node in the graph
2. For each node, create a list, which consists of the ALAP of the node itself and all its children
3. Sort these lists in an ascending order of nodes ALAP
4. Create a node list L sorted in ascending order of nodes ALAP. Use nodes sorted lists (previous 2 steps) to break a tie
5. while not the end L do
       – dequeue $v_i$ from L
       – compute earliest execution start time for $v_i$ in all processors
       – schedule $v_i$ to the processor that minimizes the node earliest execution start time
```

Fig. 3: MCP algorithm

The complexity of the MCP algorithm is $O(pv^2)$. For the DAG shown in Fig. 1, the scheduling trace of MCP algorithm is given in Table 3.

Table 3: A scheduling trace of the MCP algorithm (makespan = 85)

Step	*Selected v*	p_1	p_2	p_3	*Selected p*
1	1	0	0	0	p_1
2	2	20	28	28	p_1
3	4	40	22	22	p_2
4	5	40	37	28	p_3
5	3	40	37	33	p_3
6	6	40	37	53	p_2
7	9	41	48	53	p_1
8	8	61	44	53	p_2
9	7	61	61	53	p_3
10	10	65	69	69	p_1

3.2 Dynamic List-Scheduling Algorithms

This section briefly reviews the two main dynamic list-scheduling algorithms, which are the Earliest Time First (ETF) algorithm [9] and the Dynamic Level Scheduling (DLS) algorithm [10].

3.2.1 ETF Algorithm

The Earliest Time First (ETF) algorithm [9] computes, at each step, the earliest execution start time (EEST) for all ready nodes and selects the one with the lowest value for scheduling. The ready node is defined as the node having all its parents scheduled. When two nodes have the same value of EEST, the ETF algorithm breaks the tie by scheduling the one with the higher static level. Fig. 4 shows the ETF algorithm.

```
1. Compute the SL (static level) of each node in the graph
2. Initially, the ready nodes list includes only the entry node
3. while the ready list is not empty do
      – compute the earliest execution start time on each processor for each node in the ready list
      – select the node-processor pair that gives the earliest execution start time. Ties are broken by selecting the node with a higher SL
      – schedule the node to the corresponding selected processor
      – add the newly ready nodes to the ready list
```

Fig. 4: ETF algorithm

The complexity of the ETF algorithm is $O(pv^3)$. For the DAG shown in Fig. 1, the scheduling trace of ETF algorithm is given in Table 4.

Table 4: A scheduling trace of the ETF and DLS algorithms ($makespan = 88$)

Step	*Selected v*	p_1	p_2	p_3	*Selected p*
1	1	0	0	0	p_1
2	2	20	28	28	p_1
3	4	40	22	22	p_2
4	3	40	37	24	p_3
5	5	40	37	44	p_2
6	6	40	42	44	p_1
7	7	52	52	44	p_3
8	8	45	53	53	p_1
9	9	50	48	50	p_2
10	10	76	68	76	p_2

3.2.2 DLS Algorithm

The Dynamic Level Scheduling (DLS) algorithm [10] uses an attribute called the *dynamic level* (*DL*), which is the difference between the static level of a node and its earliest execution start time. In each scheduling step, the node-processor pair that gives the largest value of *DL* is selected. This mechanism is similar to the one used by the ETF algorithm. However, there is one subtle difference between the ETF and DLS: the ETF algorithm schedules the node with the minimum earliest execution start time and uses the static level merely to break ties. In contrast, the DLS algorithm tends to schedule nodes in descending order of their static levels at the beginning of the process, but tends to schedule nodes in ascending order of EEST near the end of the process. The DLS algorithm is shown in Fig. 5.

```
1. Compute the SL (static level) of each node in the graph
2. Initially, the ready nodes list includes only the entry node
3. while the ready list is not empty do
      – compute the earliest execution start time for every ready node on each processor
      – compute the DL of every node-processor pair by subtracting the earliest execution start time from the node's static level (SL)
      – select the node-processor pair that gives the largest DL
      – schedule the node to the corresponding selected processor
      – add the newly ready nodes to the ready list
```

Fig. 5: DLS algorithm

The complexity of the DLS algorithm is $O(pv^3)$. For the DAG shown in Fig. 1, the s of DLS algorithm is exactly the same as the scheduling trace of the the ETF algorithm as shown in Table 4.

4 Experimental Results and Discussion

This section presents a performance comparison of the four algorithms given in section 3. For this purpose, we used randomly generated task graphs and the following comparison metrics are used.

4.1 Comparison Metrics

The comparisons of the algorithms are based on the following metrics.

Makespan

The makespan is defined as the overall Completion time, and can be specified as follows:

$$makespan = FET(v_{exit}),$$

where $FET(v_{exit})$ is the finishing time of the scheduled exit node.

Scheduling Length Ratio (SLR)

The main performance measure is the scheduling length (makespan) of its output schedule. Since a large set of task graphs with different properties is used, it is necessary to normalize the schedule length to the lower bound, which is called the Schedule Length Ratio (*SLR*). The *SLR* value of an algorithm on a graph is defined as

$$SLR = \frac{makespan}{\sum_{i \in CP} w_i}.$$

The denominator is the sum of the computation costs of the tasks on a critical path (CP). The *SLR* of a graph (using any algorithm) cannot be less than one, since the denominator is the lower bound. Average *SLR* values are used in our experiments.

Speedup

The speedup value is computed by dividing the sequential execution time (i.e., the cumulative computation costs of the tasks) by the parallel execution time (i.e., the makespan of the schedule).

Number of occurrences of better quality of schedules

The number of times that each algorithm produced a better, worse, and equal quality of schedules compared to every other algorithm is counted in the experiments.

4.2 Random Graph Generator

The random graph generator was implemented to generate weighted application DAGs with various characteristics that depend on several input parameters. The generator requires the following input parameters to build weighted DAGs.

- number of tasks in a graph v,
- graph levels l,
- communication to computation ratio *CCR*, which is defined as the ratio of the average communication cost to the average computation cost.

In all experiments, graphs with a single entry and a single exit node were considered. In each experiment, the values of the previous parameters are selected from the corresponding set given below.

$$v \in \{20, 30, 40, 50, 60, 70, 80, 90, 100\},$$
$$0.2\,v \le l \le 0.8\,v,$$
$$CCR \in \{0.5, 1.0, 2.0\}.$$

These input parameters were used to generate 10*k* different DAGs with various characteristics for each v from the used v set.

4.3 Performance Results

The performances of the algorithms were compared with respect to different graph size. The experiments were repeated for each v from the v set given above. For each v, 10*k* graph were generated using random selection for *CCR* and levels (l) (given above) for each graph. The average *SLR* for each v is given in Fig. 6. In general the performances of the dynamic algorithms are better than those of the static ones. The static algorithms have near equal performance while the ETF algorithm is better than the DLS in the dynamic algorithms.

The average speedup is given in Fig. 7. In general, the speed up of the dynamic algorithms is better than the static algorithms. The two dynamic algorithms have almost the same performances, while HLFET is better than MCP.

Finally, the percentage of situations that each scheduling algorithm in the experiments produced better (B), equal (E) or worse (W) scheduling length compared to every other algorithm was counted for 90*k* DAGs used. Each cell in Table 5 indicates the comparison results of the algorithm on the left with the algorithm at the top.

Table 5 indicates that the dynamic algorithms are better than the static ones. As regards the static algorithms, the HLFET algorithm is better than the MCP, while the DLS algorithm is better than the ETF algorithm as regards the dynamic algorithms.

We note that the algorithm complexity is an important factor that has to be taken into account when comparing the performance of different algorithms. As shown in Table 6, the complexity of the dynamic algorithms is much higher than

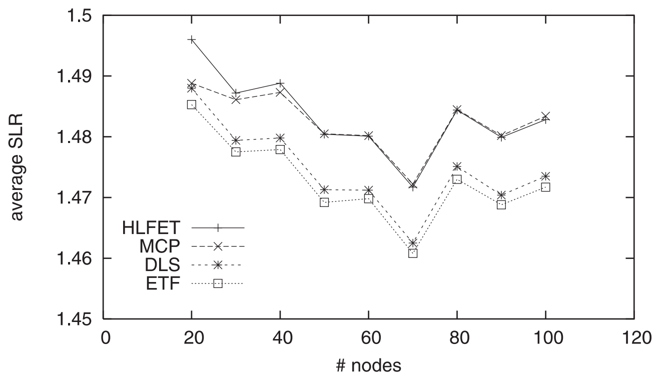

Fig. 6: Average *SLR*

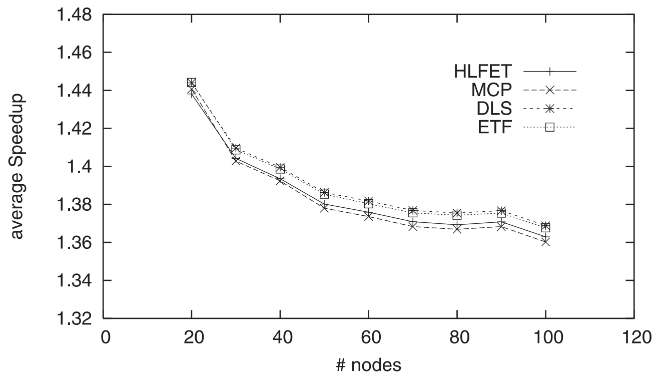

Fig. 7: Average Speedup

Table 5: Pair-Wise Comparison of the examined algorithms

		HLFET	MCP	ETF	DLS
HLFET	B		49.15 %	37.75 %	22.62 %
	E		12.47 %	22.37 %	42.17 %
	W		38.39 %	39.89 %	35.22 %
MCP	B	38.39 %		40.84 %	36.41 %
	E	12.47 %		5.84 %	8.35 %
	W	49.15 %		53.33 %	55.24 %
ETF	B	39.89 %	53.33 %		26.93 %
	E	22.37 %	5.84 %		39.53 %
	W	37.75 %	40.84 %		33.54 %
DLS	B	35.22 %	55.24 %	33.54 %	
	E	42.17 %	8.35 %	39.43 %	
	W	22.62 %	36.41 %	26.93 %	

Table 6: Algorithms complexity

Algorithm	Complexity
HLFET	$O(pv^2)$
MCP	$O(pv^2)$
ETF	$O(pv^3)$
DLS	$O(pv^3)$

that of static ones. This makes it unfair if the static algorithm gives the same schedule length as the dynamic one, to consider it as an equivalent trial. If we consider the equivalent scheduling length of two algorithms as a better trial of the lowest complexity algorithm, the better, equal, and worse comparison between the examined algorithms will be as shown in Table 7.

Table 7: Pair-Wise comparison of scheduling algorithms complexity based

		HLFET	MCP	ETF	DLS
HLFET	B		49.15 %	60.12 %	64.79 %
	E		12.47 %		
	W		38.39 %	39.89 %	35.22 %
MCP	B	38.39 %		46.68 %	44.76 %
	E	12.47 %			
	W	49.15 %		53.33 %	55.24 %
ETF	B	39.89 %	53.33 %		26.93 %
	E				39.53 %
	W	60.12 %	46.68 %		33.54 %
DLS	B	35.22 %	55.24 %	33.54 %	
	E			39.53 %	
	W	64.79 %	44.76 %	26.93 %	

4.4. Ranking of Examined Algorithms

Based on the above comparison metrics and the average results for 90*k* randomly generated DAGs, the ranking for the examined algorithms was as follows:

average Makespan:	DLS	ETF	HLFET	MCP
average Speedup:	DLS	ETF	HLFET	MCP
average SLR:	ETF	DLS	MCP	HLFET
best results:	DLS	ETF	MCP	HLFET
complexity:	HLFET & MCP		DLS & ETF	

5 Conclusion

In this paper we present a brief description of the characteristics of the two most known static and also the two most known dynamic list-scheduling algorithms. The performances of these algorithms were examined using variant random generated graphs. Six comparison matrices were used to measure their performance. In general the dynamic list-scheduling algorithms performed better than the static list-scheduling algorithms. For the static list-scheduling algorithms, the HLFET performed better than MCP and for the dynamic list-scheduling algorithms, the DLS algorithm performed better than the ETF algorithm. If the complexities of the algorithms are taken into account, it is highly recommended to use static list-scheduling.

References

[1] Feitelson D., Rudolph L., Schwiegelshohm U., Sevcik K., Wong P.: *Theory and Practice in Parallel Job Scheduling*. JSSPP, 1997, p. 1–34.

[2] Kwok Y., Ahmed I.: *Benchmarking the Task Graph Scheduling Algorithms*. Proc. IPPS/SPDP, 1998.

[3] Liou J., Palis M.: *A Comparison of General Approaches to Multiprocessor Scheduling*. Proc. Int'l Parallel Processing Symp., 1997, p. 152–156.

[4] Khan A., McCreary C., Jones M.: *A Comparison of Multiprocessor Scheduling Heuristics*. ICPP, 1994, Vol. 2, p. 243–250.

[5] Gerasoulis A., Yang T.: *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*. Journal of Parallel and Distributed Computing, 1992, Vol. 16, p. 276–291.

[6] Gerasoulis A., Yang T.: *On The Granularity and Clustering of Directed Acyclic Task Graphs*. IEEE Trans. Parallel and Distributed Systems, 1993, Vol. 4, No. 6, p. 686–701.

[7] Zhou H.: *Scheduling DAGs on a bounded Number of Processors*. Int'l Conf., Parallel and Distributed Processing Techniques and Applications, 1996.

[8] Min-You W., Gajski D.: *Hypertool: A Programming Aid for Message-Passing Systems*. IEEE Trans. Parallel and Distributed Systems, 1990, Vol. 1, No. 3.

[9] Hwang J., Chow Y., Anger E., Lee C.: *Scheduling Precedence Graphs in Systems with Interprocessor Communication Times*. SIAM Journal on Computing, 1989, Vol. 18, No. 2, p. 244–257.

[10] Sih G., Lee E.: *A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures*. IEEE Trans. In Parallel and Distributed Systems, 1993, Vol. 4, No. 2, p. 75–87.

Ing. Tarek Hagras
phone: +420 224 357 267
e-mail: tarek@felk.cvut.cz

Doc. Ing. Jan Janeček, CSc.
phone: +420 224 357 267
e-mail: janecek@fel.cvut.cz

Dept. of Computer Science and Engineering

Czech Technical University in Prague
Faculty of Electrical Engineering
Karlovo nám. 13
121 35 Prague 2, Czech Republic