Yugoslav Journal of Operations Research
10 (2000), Number 1, 123-136

EXHAUSTIVE LIST-SCHEDULING HEURISTIC FOR DENSE TASK GRAPHS*

Tatjana DAVIDOVI]

Mathematical Institute - SANU
11000 Belgrade, Yugoslavia

Abstract: The multiprocessor scheduling problem has usually been "solved" by using heuristic methods. A large number of such heuristics can be found in the literature, but none of them can efficiently be applied to the problem in its most general form. The efficiency of any scheduling heuristic depends on both the task graph structure and the multiprocessor system architecture. List-scheduling heuristics are the most common ones in use. They are based on the definition of the task priority list determining the order in which tasks will be scheduled (assigned to one of the processors). A task can be assigned to a processor in many different ways, but variations of the earliest start heuristics are the most frequent in use. In this paper a scheduling strategy based on the use of multiple task priority lists combined with several assignment heuristics is suggested in order to find optimal (near-optimal at the worst) solutions for a large class of arbitrary task graphs. It is shown that performing an exhaustive search overall feasible task priority lists is not too expensive for scheduling dense task graphs.

Keywords: Scheduling, multiprocessors, precedence graphs, communication delay.

1. INTRODUCTION

The distribution of tasks to be executed on an arbitrary multiprocessor system, often referred to as the scheduling or task allocation problem, will be discussed in this paper. The quality of the schedule is highly affected by the precedence constraints among the tasks as well as the number of processors, the topology of the communication network and the communication mechanism itself [12]. Here, it is assumed that the precedence constraints among tasks are defined by a directed acyclic graph called a dependency graph or task graph [5, 13]. The multiprocessor topology (architecture) is described by a symmetric distance matrix [1, 3].

* This research was supported by NSF Serbia, grant no. 04M02C.

The precedence relation between tasks constrains the order in which tasks can be executed to assure correctness of the parallel execution of the program. The performance of the parallel program can be radically different on various architectures, usually representing a consequence of communication and synchronization between tasks. The communication costs should not be ignored in the scheduling process: they could be higher than the computational costs.

The scheduling of parallel program modules (tasks) among processors is an NP-complete problem in its most general form [15]. A lot of papers considering this problem have appeared in recent years [1, 3-9, 11, 13, 14, 16, 17]. To avoid this problem's complexity the authors have considered restricted forms of the problem [5, 16] and/or have introduced some heuristics to find certain satisfactory suboptimal solutions [4, 6, 9, 10, 11, 13]. The restrictions are related to the selection of some fixed multiprocessor architecture [4, 6, 11, 17], the introduction of some constraints on the task graph [4, 5, 9, 16] or the assumption of constraints regarding communication delays [5, 13, 16]. Most of the heuristic approaches rely on the definition and optimization of objective functions [4, 9-11, 13] but metaheuristics such as genetic algorithms [7] and simulated anealing [14] have recently been applied, too.

The main problem with heuristics is that any particular heuristic cannot be efficiently applied to all problems. For example, if the heuristic is designed to assure load balancing of the processors constituting the multiprocessor system [11], and if there are too many processors, a lot of execution time will be spent on transferring data between processors, which would result in prolongation of the total execution time. For this reason, many researchers are working on the development of new heuristics to be efficiently applied on a larger class of problems.

The scheduling process is usually performed in two steps [3-5, 9, 11, 13]: first the task to be scheduled is selected and then the most appropriate processor for the given task is determined. This means that the tasks are ordered into some priority list and then scheduled (assigned to the processors) one by one according to that list. Heuristics based on these two steps are called list-scheduling heuristics. The order of tasks within the priority list can be defined in several ways: the length of the Critical Path[1] [13, 17], the amount of required communications [5, 11, 16], the number of successors, the task execution time, the task mobility [3, 4, 6], etc. Some of these strategies obey precedence constraints between tasks [4, 5, 13, 16] and the others do not (precedence constraints are considered only in the second scheduling step). The assignment of tasks to the processors (second scheduling step) can also be performed by using different criteria: the earliest start of the task [4, 13], minimum communications [5, 11, 16], minimal prolongation of task graph execution time [3, 6], etc.

In this paper, it is proposed to use more than one priority list to schedule the task graph, as well as to apply more than one assignment strategy forcing minimization of both execution and communication times. A number of so-called "feasible lists" is

[1] Here CP is defined as the largest sum of execution times till an endtask.

generated and scheduled in two different ways to be executed on the given multiprocessor system. The best obtained schedule is taken for the final one. A "feasible list" of tasks represents any permutation of tasks which obeys the precedence constraints defined by the task graph. The scheduling process proposed in this paper follows the precedence relations between tasks. The number of all the feasible lists depends reversely upon the number of task graph edges (except in some special cases as descussed in Section 3). Having this in mind it is proposed to examine all feasible lists for dense task graphs and to take the best obtained schedule for the final one. In the case of scheduling sparse task graphs, it is suggested to perform the scheduling process (generation of feasible lists and assignment of tasks directed by these lists) within a given time limit or until some other criterion is satisfied. For assigning tasks to processors the well known "earliest start" heuristic is used, as well as declustering method experimentally proven to be very efficient for scheduling sparse task graphs.

The paper is organised as follows: the definition of the scheduling problem is given in Section 2, Section 3 contains a description of the proposed scheduling method, examples and experimental results are given in Section 4, while Section 5 concludes this paper.

2 PROBLEM STATEMENT

Scheduling parallel programs to be executed on multiprocessor systems may be static or dynamic [12]. Static scheduling means a priori assignment of tasks to the processors (this assignment does not change during parallel program execution). Dynamic scheduling [2] represents a run-time assignment and it is performed when the task graph structure changes during the scheduling process. In this paper static scheduling is considered. Starting from both the program which is to be parallelized and a given multiprocessor architecture, the problem is to determine *where* and *when* each program module (task) will be executed.

The tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple $G=(T,E,C,L)$ where $T=\{t_1,\dots,t_n\}$ represents the set of tasks; $E=\{e_{ij}\mid t_i,t_j\in T\}$ represents the set of communication edges; $C=\{c_{ij}\mid e_{ij}\in E\}$ represents the set of edge communication costs; and $L=\{l_1,\dots,l_n\}$ represents the set of task computation times (lengths). The communication cost $c_{ij}\in C$ defines the amount of data transferred between tasks t_i and t_j if they are executed on different processors. If both tasks are scheduled to the same processor, the communication cost is zero. The set E defines precedence relations between tasks. A particular task cannot be executed unless all of its predecessors complete their own execution and all relevant data are available. Task preemption and redundant execution are not allowed.

It is assumed that the multiprocessor architecture M contains p identical processors with their own local memories. The processors, exchanging messages, communicate through bidirectional links. This architecture can be modeled by a processor graph [10] or by a distance matrix [1]. The nodes of a processor graph represent processors, while links between processors are modelled by the graph edges. The element (i, j) of the distance matrix $D = [d_{ij}]_{p \times p}$ is equal to the minimal distance between the nodes i and j. Here, the minimal distance is equal to the number of links along the shortest path between two nodes. It is easy to see that the distance matrix is symmetric having zero diagonal elements. It is also assumed that the processors are connected with links of the same capacity.

The scheduling of DAG G to be executed on M consists of the determination of the index of the associated processor, and the calculation of the starting time for each of the tasks from the task graph, to minimize some objective function. The usual objective function can be the execution time of the scheduled task graph (makespan, schedule length, used in this paper as well) [13], but it can also be the load imbalance of the processors [11] or the total communication cost [9].

3. SCHEDULING HEURISTIC

3.1. Description of the scheduling heuristic

The precedence relation defines a partial order between tasks, i.e. there are some independent tasks which can be executed simultaneously, meaning that these tasks can be scheduled in an arbitrary order. Thus, the first step in the list-scheduling method is to determine priorities for all the tasks for obtaining a total order between them. The obtained list of tasks defines the order for task scheduling. A task is ready for scheduling if all of its predecessors have already been scheduled.

The next step in the scheduling process usually is to select the most appropriate processor on which the given ready task will be executed. All the necessary communication routines must be added, too. Finally, the task starting time is calculated. This structure of the scheduling heuristic can be found in [4, 9, 10, 13]. Several recently developed heuristics [3, 5, 6, 9, 11] have not followed precedence constraints in the process of generating task priority lists. This caused an increase in computational complexity because within the assignment part of the scheduling process, task starting times are not known at once.

The heuristic proposed in this paper is also based on the list-scheduling method. It takes into consideration all lists obeying the precedence constraints among tasks (from now on they will be referred to as feasible lists*)*. The set of such lists is a subset of all the permutation of tasks, and the number of lists in that subset depends on the task graph structure (number of tasks and number of communication edges): while the task graph complexity increases, the number of feasible lists decreases. Yet, it is obvious (see next subsection) that the number of feasible lists cannot be polynomially limited w.r.t. the number of tasks and/or edges in a task graph, in general.

The i-th feasible list can be described as a permutation $(t_{i1},\dots,t_{ik},\dots,t_{il},\dots,t_{in})$ of all the tasks where $t_{ik},\dots,t_{il}$ means that either tasks t_{ik} and t_{il} are independent or there is a path from t_{ik} to t_{il} with all edges in E.

The procedure for finding all feasible lists (permutations of tasks) is recursive (Fig. 1) and can be described as follows.

```
i = 0;
IRT = list of initially ready tasks;
while $(IRT \neq \emptyset)$ {
    task = next(IRT);
    fea_perm[i] = task;
    NRT = IRT \ {task} + succ(task)
    all_fea(NRT, i + 1);
}
```

Figure 1: Procedure for generation of all feasible lists of tasks

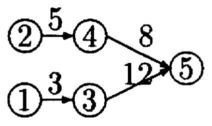
Tasks with no predecessors are included in the list of initially ready tasks (IRT). The tasks are enumerated to define order between them. A recursive procedure (all_fea), for finding all feasible permutations with task (task $\in$ IRT) as their first element, is called for each initially ready task. Within this procedure the ordered list of new ready tasks (NRT) is generated by including all the remaining ready tasks $(IRT \setminus \{task\})$ as well as the new ones which are becoming ready because all of their predecessors are already included in the feasible list[2]. Having NRT empty, all_fea calls the assignment procedure to schedule tasks to the processors.

In this paper it is suggested to use two assignment procedures, but there are no restrictions to use any other heuristics based on the list scheduling method with priority lists designed to obey precedence constraints between tasks. The result of the assignment procedure (schedule length) is compared with the current minimum, and if it is better (smaller) it is saved as the new minimum (together with the corresponding schedule, defined by the index of the associated processor and the starting time for each of the tasks).

The first assignment procedure used in this paper is the well known Earliest Start (ES) method: given ready task (t) is scheduled as early as possible. The starting time of task t is calculated for all the processors, depending upon the communication times with predecessors scheduled on other processors and the current ocupancy of a given processor. Minimum starting time defines the processor j on which task t will be executed. Although proven to be very efficient and widely used, ES heuristics can give

[2] Here, the definition of ready task is changed because the feasible list generation and assignment process are separated.

unsatisfactory solutions [8], sometimes even with execution times longer than the serial execution time of a given task graph, as is the case in the following example.



t_i	1	2	3	4	5
l_i	2	3	5	8	4

a)

	feasible lists					schedule length
1	1	2	3	4	5	23
2	1	2	4	3	5	23
3	1	3	2	4	5	23
4	2	1	3	4	5	23
5	2	1	4	3	5	23
6	2	4	1	3	5	23

b)

Figure 2: a) Task graph example; b) Corresponding ES schedules

The task graph presented on Fig. 2a cannot be optimally scheduled on two connected processors by the use of ES heuristic. Six feasible lists yield to the same schedule length 23 (Fig. 2b), while the sequential execution time of this task graph equals 22. This is because of the large amount of required communication and the lack of tasks for filling the idle interval. One of the ES obtained schedules and the optimal ones are given as follows:

Heuristic schedule				Optimal schedules									
P_1 :	1_2	3_7		P_1 :	1_2				P_1 :	1_2	3_7	${}_9 4_{16}$	5_{20}
P_2 :	2_3	4_{11}	${}_{20}5_{23}$	P_2 :	2_3	4_{11}	3_{16}	5_{20}	P_2 :	2_3			

where the back task index represents the ending time of a given task, while the front one (if any) denotes the starting time of a given task (if it is different from the ending time of the preceeding task).

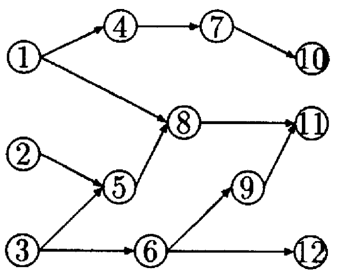
As can be seen from the above example, sometimes it is better to delay task execution than to wait for its results.

This indicates that more than one heuristic should be used to assure the efficient scheduling of similar task graph examples. Since none of the tested heuristics [5, 9-11, 13] gave the optimal solution for the example on Fig. 2, in this paper a variant of the (de)clustering method which will be called DC is used. An initial serial schedule defined by a feasible list, obtained by assigning all the tasks to one of the processors (P_1) , has to be improved by moving tasks one by one to another processor (following the task order defined by that feasible list). Each task t in the feasible list is scheduled to all the remaining processors and finally moved to another processor P_j so that the total execution time of the entire task graph is maximally reduced. When task t is moved to some other processor P_i the new starting times for all the remaining tasks in the feasible list have to be calculated: there is a free space left on P_1 but new communications between P_1 and P_i have to be added. If no improvement is made (total execution time does not decrease if t is moved to any of the remaining processors), task t is being returned to the first processor. The task graph given on Fig. 2 can be optimally scheduled by using the DC assignment strategy. For example, the second given optimal schedule can be obtained starting from the permutation 21345 and just moving task 2 to processor P_2 .

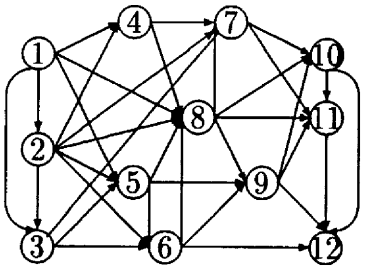
A comparison of these two assignment strategies is given in Section 4.

3.2. Complexity of the proposed heuristic

The complexity of the second step of the scheduling heuristic proposed in this paper is $O(n^2 p)$ for the ES assignment heuristic and $O(n^4 p)$ if DC is used. This is because the starting time of each task (O(n)) on every processor (O(p)) has to be calculated having in mind where the task predecessors have been scheduled (O(n) at the worst). While using DC, the new starting times for the remaining tasks in the feasible list have to be calculated, too $(O(n^2))$. Henceforth, the entire heuristic complexity depends on how many times this second step is performed, i.e. on the number of feasible lists of tasks.



a)



b)

Figure 3: a) Sparse task graph b) Modified task graph

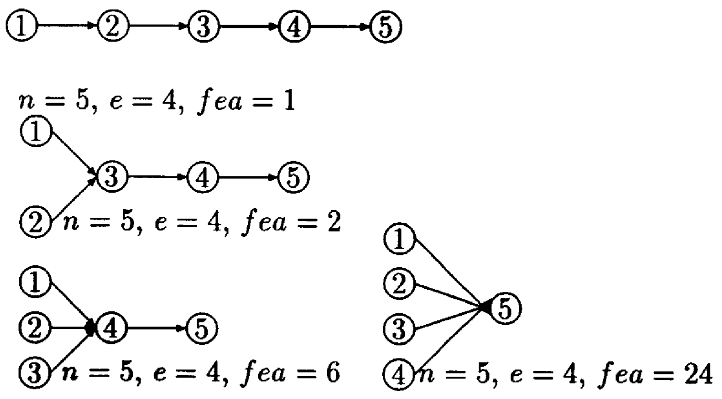
To illustrate how the number of feasible lists depends on task graph complexity (number of communication edges), let us consider the following example (Fig. 3a).

The task graph contains n = 12 nodes (tasks) and e = 12 edges (e = | E|). The number of feasible lists equals fea = 36130. The density of this task graph ρ = 18.18%. Table 1 shows how the number of feasible lists decreases while adding new communication edges to the task graph on Fig. 3a. When the number of communication edges is set to 34 (ρ = 51.51%) only eight feasible lists of tasks will remain. The modified task graph is presented on Fig. 3b.

Table 1: Number of feasible lists depending on task graph density

e	12	13	15	16	18	19	21	23	25	28	30	31	32	34
fea	36130	32198	19042	5426	4026	1524	1068	704	616	280	112	56	28	8

Fig. 4 presents a special case example to illustrate that the dependency of the number of feasible lists on the number of task graph edges cannot be expressed by a simple formula. All the task graphs from that figure contain the same number of nodes and edges, yet the number of feasible lists is quite different.



$n = 5,\ e = 4,\ fea = 1$
$n = 5,\ e = 4,\ fea = 2$
$n = 5,\ e = 4,\ fea = 6$
$n = 5,\ e = 4,\ fea = 24$

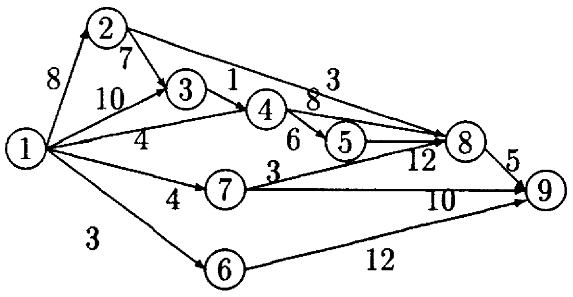
Figure 4: Task graphs representing special cases

Since the number of feasible lists is too large for sparse task graphs, the process of generating of all the feasible lists can be limited by some stopping criterion (execution time of the scheduling process, percentage of improvement made, etc.). Some examples of scheduling results, obtained within a given time limit, are presented in the next section.

Several cut rules are used to reduce computational complexity. Any partial schedule longer than the current minimum is not for further consideration. Only neighbor processors and those containing predecessors [6] are considered. In addition, any moment the obtained schedule length is equal to the length of the critical path, the scheduling process is stopped because the optimal schedule has been found.

4. EXAMPLES AND EXPERIMENTAL RESULTS

An illustrative example of a task graph is given in Fig. 5 where it is assumed that the multiprocessor system contains two identical processors (according to the task graph structure it is pointless to use more than two processors).



t_i	1	2	3	4	5	6	7	8	9
l_i	60	30	15	40	15	30	35	50	40

Figure 5: An example of a task graph

The task graph contains 9 tasks with non-uniform execution and communication times. The length of the Critical Path equals 250 representing the minimal value for the length of the optimal schedule. The scheduling results are presented in Table 2. The total number of feasible lists equals 35, and all the schedules obtained using both assignment heuristics are listed in Table 2. The minimal schedule length obtained equals 250 and there are 22 possible optimal solutions when ES is applied, and only 6 if the DC assignment strategy is used. It can be seen from Table 2 that if the priority list is not properly defined, unsatisfactory solutions will be obtained (about 13% worse than the optimal one for ES and 18% for DC). On the contrary, if the task priority is defined as the largest sum of execution times till an endtask (see 7th row in Table 2) and ES is used in the second scheduling step, the optimal schedule will be obtained within $O(n^2 p)$ time.

The proposed scheduling process was tested on random task graphs with non-uniform distribution for both execution times and communication costs. Task graph density varied from 40% to 80%. The obtained schedules were compared with other heuristics [5, 9-11, 13]. For all tested task graphs the DLS heuristic proposed in [13] is performed better than the other ones. For that reason, comparision results are only given for schedules obtained by DLS and the scheduling heuristic proposed in this paper. Some illustrative scheduling results are summarized in Table 3.

Table 2: Feasible schedules for the task graph given in Fig. 5

	feasible lists									ES	DC
1	1	2	3	4	5	6	7	8	9	250	270
2	1	2	3	4	5	7	6	8	9	250	294
3	1	2	3	4	5	7	8	6	9	250	262
4	1	2	3	4	6	5	7	8	9	250	263
5	1	2	3	4	6	7	5	8	9	250	257
6	1	2	3	4	7	5	6	8	9	250	251
7	1	2	3	4	7	5	8	6	9	250	273
8	1	2	3	4	7	6	5	8	9	250	274
9	l	2	3	6	4	5	7	8	9	250	278
10	1	2	3	6	4	7	5	8	9	250	257
11	1	2	3	6	7	4	5	8	9	250	281
12	1	2	3	7	4	5	6	8	9	250	264
13	1	2	3	7	4	5	8	6	9	250	297
14	1	2	3	7	4	6	5	8	9	250	258
15	1	2	3	7	6	4	5	8	9	250	286
16	1	2	6	3	4	5	7	8	9	250	265
17	1	2	6	3	4	7	5	8	9	250	263
18	1	2	6	3	7	4	5	8	9	250	258
19	1	2	6	7	3	4	5	8	9	257	250
20	1	2	7	3	4	5	6	8	9	250	265
21	1	2	7	3	4	5	8	6	9	250	262
22	1	2	7	3	4	6	5	8	9	250	257
23	1	2	7	3	6	4	5	8	9	250	260
24	1	2	7	6	3	4	5	8	9	259	250
25	1	6	2	3	4	5	7	8	9	258	256
26	1	6	2	3	4	7	5	8	9	258	257
27	1	6	2	3	7	4	5	8	9	258	258
28	1	6	2	7	3	4	5	8	9	258	250
29	1	6	7	2	3	4	5	8	9	280	250
30	1	7	2	3	4	5	6	8	9	258	256
31	1	7	2	3	4	5	8	6	9	258	297
32	1	7	2	3	4	6	5	8	9	258	257
33	1	7	2	3	6	4	5	8	9	258	260
34	1	7	2	6	3	4	5	8	9	258	250
35	1	7	6	2	3	4	5	8	9	283	250

Table 3: Random task graph scheduling results

			schedule			CPU time [sec]	
N	ρ	p	DLS	ES	DC	ES	DC
20	80	2	338	325	325	0.02	0.1
20	70	2	312	292	292	0.03	0.22
20	60	2	280	271	265	0.28	0.23
20	50	2	289	265	249	0.10	1.07
20	40	2	213	197	191	1837.82 (≈0.5h)	1047.82
20	80	4	338	325	325	0.02	0.17
20	70	4	312	292	292	0.05	0.3
20	60	4	292	272	255	0.49	4.34
20	50	4	288	271	248	0.25	1.93
20	40	4	201	197	191	3594.87 (< 1h)	2079.79
50	80	2	732	725	727	0.64	9.02
50	70	2	765	747	759	6.28	26.98
50	60	2	798	762	768	4623.30 (< 1.25h)	158268.42
50	50	2	650	618	*	64789.56 (≈18h)	*
50	80	4	732	725	727	1.33	13.95
50	70	4	465	747	759	12.08	47.80
50	60	4	798	760	768	9404.43 (≈2.61h)	281988.4
50	50	4	650	618	**	130642.49 (< 37h)	**
100	80	2	1658	1612	1640	8848.09 (≈2.45h)	129663.20
100	70	2	1621	1568	*	82769.24 (≈23h)	*
100	80	4	1658	1612	**	18104.36 (≈5h)	**
100	70	4	1629	1572	**	162022.87 (< 45h)	**

* - the results are missing because long execution time is required;

** - it is pointless to use four processors in these examples.

The first two columns describe the task graph structure (n represents the number of tasks and ρ denotes the density of graph edges). The number of processors within multiprocessor system p is given in the third column (p = 2 means a multiprocessor system with two connected processors, while p = 4 denotes a 2-dimensional hypercube). Schedule lengths obtained using DLS, ES, and DC heuristics are presented in the next three columns respectively, while the last two columns contain CPU time spent by ES and DC heuristics, respectively. CPU time needed for DLS heuristic execution is polynomial (related to the number of nodes in the task graph and the number of processors - $O(n^3 p)$), and it can be neglected. The values of CPU time are given for a Pentium PRO microprocessor with the Linux operating system. Judging by this table, we can conclude that good schedules for dense task graphs can be obtained in a reasonably short time. The large density of the tested task graphs is the reason for using such a small number of processors (2, 4). If a lot of processors are used, the required intensive data transfer between tasks, scheduled to be executed on different processors, would prolong the total execution time of the task graph. Comparing rows 3 and 8, as well as rows 4 and 9 from Table 3, it can be seen that a multiprocessor system with only two processors executes the task graph scheduled by the ES heuristic faster than the 2-dimensional hypercube. The DC heuristic is designed to avoid to some extent such diversifications of tasks among the processors. As can be seen from the Table 3 the proposed exhaustive search overall feasible permutations of tasks ensures that obtained schedule is better than DLS result, but with sacrifice of the polynomial computational complexity.

Table 4: Time limited scheduling results

			CPU=40 sec		CPU=l0 min		CPU=1h	
n	ρ	p	ES	DC	ES	DC	ES	DC
20	40	2	197	191	197	191	197	191
20	30	2	213	215	204	209	204	205
50	50	2	630	657	630	657	630	657
50	40	2	734	730	734	730	730	721
50	30	2	539	506	514	501	509	475
100	50	2	1674	1647	1670	1647	1652	1647
100	40	2	1512	1534	1512	1534	1507	1525
100	30	2	1205	1312	1205	1294	1196	1292

To avoid computational complexity problem in sparse task graph scheduling, these two heuristics are compared within a given CPU time limit. The obtained comparison results are given in Table 4. As it can be seen from Tables 3 and 4, the use of different assignment strategies can significantly improve scheduling results. This improvement is evident even if the execution time of the scheduling process is limited.

Beside these two assignment heuristics (ES and DC), any other known list-scheduling heuristic with priority lists designed to obey percedence constraints ([13, 17] for example), or some new ones, can be used in order to improve the final scheduling result: minimization of the total execution time of a given task graph. The procedure for generating and scheduling feasible lists can easily be parallelized in order to speed up its execution. It is also possible to define some local search procedure based on examining feasible lists of tasks, in order to direct search for minimal schedule length. A parallel genetic search algorithm, where members of the population are represented as feasible lists of tasks, has already been developed [7].

The description of the class of all the task graphs that can be scheduled optimally using the heuristic proposed in this paper does not follow straightforward. Detailed analysis is needed to find the conditions that a certain task graph should satisfy, assuring that its optimal schedule is defined by one of the feasible lists.

5. CONCLUSION AND FUTURE WORK

An exhaustive list-scheduling heuristic for multiprocessor task scheduling has been proposed. The heuristic has been based on the modification of a well known list-scheduling method. All lists of tasks obeying precedence constraints have been determined, a schedule has been performed for each list using two assignment heuristics, and the best schedule has been selected as the final one. The complexity of the proposed heuristic depends on the task graph structure, and this complexity decreases while graph density increases within the same dimension, i.e. number of tasks. It has been experimentally proven that an exhaustive search overall feasible lists of tasks was not too expensive for dense task graphs with number of tasks not greater than 100, while for sparse ones it has been recommended to use a time (or some other criterion) limited scheduling process. The optimality of the obtained schedule cannot be guaranteed in the general case.

The non-polynomial computational complexity of the proposed scheduling heuristic as well as the problem of obtaining optimal solutions direct further investigations: a) to parallelization of the scheduling process, b) to developing local search procedures based on examining feasible permutations, and c) to searching for conditions that task graphs should satisfy in order to be scheduled optimally by this exhaustive list-scheduling procedure.

REFERENCES

[1] Davidovi}, T., "An efficient multiprocessor task scheduling", Proceedings of Yugoslav Symposium on Operations Research, Herceg-Novi, 1998. (in Serbian).

[2] Dertouzos, M.L., and Mok, A.K., "Multiprocessor on-line scheduling of hard-real-time tasks", IEEE Trans. on Software Engineering, 15 (12) (1989) 1497-1506.

[3] Djordjevi}, G., and To{i}, M., "A compile-time scheduling heuristic for multiprocessor architectures", The Computer Journal, 39 (8) (1996) 663-674.

[4] Kir}anski, N., Davidovi}, T., and Vukobratovi}, M., "A contribution to parallelization of symbolic robot models", Robotica, 13 (1995) 411-421.

[5] Krishnamoorthy, V., and Efe, K., "Task scheduling with and without communication delays: A unified approach", European Journal of Operational Research, 89 (1996) 366-379.

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

[7] Kwok, Y.-K., and Ahmad, I., "Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm", J. Parallel and Distributed Computing, 47 (1997) 58-77.

[8] Kwok, Y.-K., and Ahmad, I., "Fastest: A practical low-complexity algorithm for compile-time assignment of parallel programs to multiprocessors", IEEE Trans. on Parallel and Distributed Systems, 10 (2) (1999) 147-159.

[9] Malloy, B.A., Lloyd, E.L., and Soffa, M.L., "Scheduling DAG's for asynchronous multiprocessor execution", IEEE Trans. on Parallel and Distributed Systems, 5 (5) (1994) 498-508.

[10] Manoharan, S., and Thanisch, P., "Assigning dependency graphs onto processor networks", Parallel Computing, 17 (1991) 63-73.

[11] Sarje, A.K., and Sagar, G., "Heuristic model for task allocation in distributed computer systems", IEE Proceedings-E, 138 (5) (1991) 313-318.

[12] Sarkar, V., Partitioning and Scheduling Parallel Programs for Multiprocessors, The M.I.T Press, Cambridge, MA, 1989.

[13] Sih, G.C., and Lee, E.A., "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures", IEEE Trans. on Parallel and Distributed Systems, 4 (2) (1993) 175-187.

[14] Sofianopoulou, S., "Assignment of distributed processing software: A comparative study", Yugoslav Journal of Operations Research, 7 (2) (1997) 247-255.

[15] Ullman, J.D., "NP-complete scheduling problems", J. Comput. Syst. Sci., 10 (3) (1975) 384-393.

[16] Varvarigou, T.A., Roychowdhury, V.P., Kailath, T., and Lawler, E., "Scheduling in and out forests in the presence of communication delays", IEEE Trans. on Parallel and Distributed Systems, 7 (10) (1996) 1065-1074.

[17] Yang, T., and Gerasioulis, A., "Dsc: Scheduling parallel tasks on an unbounded number of processors".