Heterogeneous Multiprocessor Scheduling with Differential Evolution

Krzysztof Rzadca, Seredynski Franciszek

Heterogeneous Multiprocessor Scheduling with Differential Evolution

Krzysztof Rzadca*
Polish-Japanese Institute of Information Technology,
Koszykowa 86,
02-008 Warsaw, Poland
krz@pjwstk.edu.pl
and
ID-IMAG,
51 avenue Jean Kuntzmann,
38330 Montbonnot Saint Martin, France

Franciszek Seredynski
Institute of Computer Science,
Polish Academy of Sciences,
Ordona 21, 01-237 Warsaw, Poland
sered@ipipan.waw.pl
and
Polish-Japanese Institute of Information Technology,
Koszykowa 86,
02-008 Warsaw, Poland

Abstract- The problem of scheduling a parallel program given by a Directed Acyclic Graph (DAG) of tasks is a well-studied area. We present a new approach which employs Differential Evolution to numerically optimize the priorities of tasks. Our algorithm starts with a number of acceptable solutions, results of different heuristics, and merges them to achieve better one in a small number of function evaluations. The algorithm outperforms both a number of greedy heuristics and a classical genetic algorithm on the most of the program graphs considered in our experiments.

1 Introduction

Parallel computing seems to be the only feasible solution for delivering the computational power required by many research and industry projects. However, employing a parallel computer instead of a sequential one results in a number of new challenges. Mapping individual tasks to the available resources and scheduling the order of execution is one of the hardest issues. At the same time it is crucial for obtaining the optimum performance of the system.

In general, a parallel program can be divided into individual tasks. Some tasks cannot be computed until they have received input data, which can be the result of another tasks. Such inter-task dependencies lead naturally to the construction of a DAG of a program. The problem of DAG scheduling can be defined as the problem of assignement of tasks to processors and ordering them with the aim of minimizing the execution time (makespan) of the whole program. This problem, except some bounded conditions, is NP-hard [4].

There were many attempts [6] to use global search meta-heuristics, such as genetic algorithms (GA), simulated annealing, tabu search, ant colonies, immune systems, or cellular automata for the problem of DAG scheduling (see [10] for an overview). Generally, such methods can be applied in one of the following ways:

- *preprocessing*, which modifies the DAG, such as clustering [5][3] of individual tasks into groups. Sometimes those algorithms also perform allocation or scheduling;
- *allocation* [8], that is decision making at which processor each task should be executed. Given the allocation, a schedule for each processor is produced by a list scheduling algorithm;
- *ordering* [1] [2], i.e. determining the order in which the tasks will be executed. The allocation depends on a list scheduling algorithm;
- *ordering and allocation*[10].

Because the preprocessing modifies the DAG, it is out of the scope of this paper. List scheduling algorithms used by the algorithms in the second and the third approaches will be described later in this paper.

There were many previous contributions on optimizing allocations, which seems to be the simplest approach to solve the DAG scheduling problem. In this case there are no unfeasible solutions. However, a typicaly used solution encoding (e.g. in a GA the value of kth gene represents a processor where kth task is executed) could result in an increased complexity of the algorithm. In the case of a GA and a 2 processor system, we obtain the typical, binary-encoded chromosomes. Unfortunately, if the number of processors increases, we are leaving the well-explored area, which in turn makes some previously suggested approaches less effective.

The optimization of the ordering was covered briefly in the literature. One can look at this problem either as an optimal ordering problem, and apply techniques similar to those used in traveling salesman problem, or as a numerical optimization of values of priorities of tasks. In later case the priorities lead to the ordering. The second approach seems to be an interesting solution, as some degree of "fuzziness" can be introduced. Think of a typical crossover operator in a GA and two individuals I_1 and I_2 which encode a different order of tasks t_1 and t_2. In ordering-like encoding, a child would have, randomly, either t_1 or t_2 executed first. In the numerical encoding, the difference between the priorities of t_1 and t_2 expresses the "degree of certitude", i.e. how certain is the individual that e.g. t_1 should be executed before t_2. If I_1 is "sure" that t_1 should be executed before t_2 (the difference between the priorities is significant), and I_2 is "uncertain" of the opposite (the difference is insignificant), the child would most probably have t_1 executed before t_2. On the other

*Krzysztof Rzadca is partly supported by the French Government Grant number 20045874

hand, if both I_1 and I_2 are "sure" of their orderings, the child should be "uncertain" of the resulting ordering. To our best knowledge the numerical optimization of priorities was used only in [1] [2], where a simple GA was used to evolve integer-encoded chromosomes. In this approach the value of kth gene gives the priority of kth task. The typical one-point crossover and re-initializing mutation are utilized as genetic operators.

The forth approach, optimizing both the ordering and the allocation, can potentially lead to the best results, as it gives the greatest "freedom" to the global optimization method. There is no greedy algorithm which could "spoil" its solutions. On the other hand, the search space is very large, hence those algorithms are expected to be rather costly. In [10] an optimal solution to scheduling DAGs with less than 30 nodes required 600–1600 generations of a GA with population size 400.

Our algorithm applies the concept of the Differential Evolution (DE) [7] to optimize the priorities of tasks. By employing a numerical optimization technique it is able to explore inter-tasks dependencies at the level much finer (i.e. fuzzy-like) than the optimal ordering approaches. By means of the DE, an optimization scheme acting on real-valued chromosomes, the genetic operators act very close to the problem domain, therefore they search through the space of possible solutions efficiently. In addition, the algorithm starts with a number of "acceptable" solutions, which are results of different greedy heuristics, allowing the search process to start in "promising" regions.

The rest of this paper is organized as follows. Section 2 contains a detailed description of the scheduling problem which we consider in this paper. In Section 3 we present a generic framework of a typical list scheduling algorithm which is the base for our approach. Section 4 contains a brief introduction to the Differential Evolution. Section 5 presents the Differential Scheduler – our scheduling algorithm. Results of conducted experiments are described in Section 6. Last section concludes the paper and presents directions for future work.

2 DAG Scheduling Problem

The system architecture on which the program is scheduled is given by a system graph, $G_s = (P, E_s)$ (Figure 1 a). The set $P = \{p\}$ of $M \equiv |P|$ nodes represents the processing units (which can be individual physical processors, workstations, or clusters), later called processors. The edges E_s represent the connections between processors. The distance $dist(p_i, p_j)$ between two processors p_i and p_j is the length of the shortest path (the one which contains the minimal number of edges) between the nodes p_i and p_j in the graph G_s. We assume fully heterogeneous model with unrelated processors. The time needed for computing the ith task (w_i^m) depends both on the task t_i and the processor p_m.

The parallel program to be scheduled is represented by a DAG, $G_p = (T, E_p)$, where $T = \{t\}$ is a set of $N \equiv |T|$ nodes which represent individual, indivisible tasks of the program (Figure 1 b). $E_p = \{e_{i,j}\}$ is the set of edges which

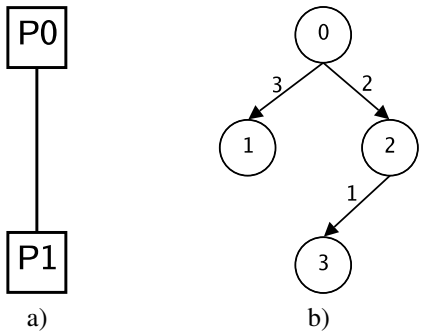


Figure 1: Example graphs of a two-processor system (a) and a four-task program (b).

represents dependencies and communication between tasks. If the tasks t_i and t_j are connected by an edge $e_{i,j}$, the task t_j cannot start until t_i completes and the data (the result of task t_i) is received. During the execution of the program a task t_j is called a *ready* task, if all such t_i are completed. By $succ(t_i)$ we denote the set of immediate successors of t_i, similarly $pred(t_i)$ is the set of immediate predecessors of t_i. The weight of an edge $e_{i,j}$ defines the time necessary to send the data from the task t_i to t_j when those two tasks execute on neighboring processors. If those tasks execute on the same processor p we assume that the time $c_{i,j}^{p,p}$ needed for the communication is zero. Otherwise, the time $c_{i,j}^{p,q}$ needed for communication is a product of the edge's weight and the distance between processors: $c_{i,j}^{p,q} = e_{i,j} * dist(p_p, p_q)$.

An important parameter of a parallel program is the mean Communication to Computation Ratio (CCR). It can be defined as the average of weights of edges divided by the average of tasks' computation times.

We assume no task duplication. Each task must be executed on exactly one processor. The tasks are not preemptive, i.e. once the execution of a task has started, it cannot be interrupted until it completes.

The DAG scheduling problem we consider in this paper is the minimization of the total execution time (the makespan) of the program taking into account the constraints defined by the DAG and the communication costs resulting from the system architecture and non-zero weights of the edges.

3 List Schedulers for Heterogeneous Model

List scheduling is one of the most popular heuristic approaches to the problem of DAG scheduling. One can divide a typical list scheduler algorithm (such as HEFT [9]) into two steps: priority computation heuristic (sometimes referred as a policy) and scheduling. In the first step, a number specifing the task's priority is associated with each task. In the second step, the tasks are ordered by decreasing priority and each task is allocated to the processor which gives the minimum completion time. Those two steps will

be covered in the next subsections.

3.1 Priority Computation

Priority computation can be viewed as the aggregation of the information about DAG structure in each node. The simplest way is to define the priority $pr(t_i)$ of a task t_i recursively [11], either by so-called upward ranking $r_u(t_i)$:

$$r_u(t_i) = \max_{t_j \in succ(t_i)} \Big(r_u(t_j) + comm(c_{i,j}^{0,0}, \ldots, c_{i,j}^{m,m'}, \ldots, c_{i,j}^{M,M}) \Big) + comp(w_i^1, \ldots, w_i^m, \ldots, w_i^M), \quad (1)$$

or by downward ranking $r_d(t_i)$:

$$r_d(t_i) = \max_{t_j \in pred(t_i)} \Big(r_d(t_j) + comp(w_j^1, \ldots, w_j^m, \ldots, w_j^M) + comm(c_{j,i}^{0,0}, \ldots, c_{j,i}^{m,m'}, \ldots, c_{j,i}^{M,M}) \Big), \quad (2)$$

where $comp$ is a function which aggregates the times needed for computation on individual processors, $comm$ is a function which aggregates the time needed for communication, w_i^m gives the time needed for computation of task t_i on processor p_m, $c_{i,j}^{m,m'} = dist(p_m, p_{m'}) \cdot e_{i,j}$ is the time needed for the communication between tasks t_i and t_j, when those two tasks are executed on processors m and m' respectively. There might be other approaches refining the formulas (1) and (2), which consider e.g. the critical path in the DAG.

In the homogeneous processor model, the value of $comp$ depends only on task. However, in heterogeneous model computation time depends also on the processor. Therefore $comp$ must somehow aggregate information about different processing times. The algorithm also lacks information about the exact communication time. As during this phase the allocation of tasks to processors is unknown, the heuristic does not know the distance between the processors on which two dependent task will be executed. Consequently, $comm$ has to aggregate information about every possible communication time. Six possible functions for $comp$ and $comm$ were defined in [11]:

- *mean* – the average of input arguments,
- *median* – the median of input arguments,
- *simple best* – the minimum of input arguments,,
- *best* – for $comp$, returns the minimum computation time, for $comm$ returns the time needed for communication on two processors which give minimum computation cost for the dependent tasks,
- *simple worst* – the maximum,
- *worst* – like *best*, but takes maximum values.

The biggest problem is that, given a DAG, it is hard to judge a priori which function will yield the best result. In our experiments, two well-performing combinations were upward ranking (Eq. 1) with *mean* function (best makespan in 28% of experiments) and upward ranking with *best* function (best makespan also in 28%). However, the worst functions had score of 10% (note that for one experiment more than one function can deliver the best makespan, so those values do not have to sum up to 100%). We observed no correlation between the performance of heuristics and the graph type, nor the number of nodes, nor the CCR.

3.2 Scheduling

In this step, the list scheduling algorithm assigns tasks to processors and determines the order of execution. There are two possible approaches to this issue. The scheduler can use tasks' priorities (or the order suggested by the global search algorithm) either as hints for tie-breaking when two or more ready tasks compete for processors or as the order in which tasks will be assigned to processors.

The first case is computationally more expensive. An ordered list of ready tasks must be maintained. Each time a task completes, this list must be updated and some new tasks must be scheduled for execution. This requires inserting temporary each ready task into the schedule of each processor (in the heterogeneous model the processor which gives the earliest time of completion is not necessarily the one which is free in the moment of scheduling) and computing the resulting start and completion times. In order to follow the priorities given by the algorithm as closely as possible, the algorithm should not schedule the tasks which would start later than the next scheduling step (when the next task will complete). Otherwise, in the next scheduling step there may be a ready task with priority higher than the priority of one of the tasks already scheduled, but not started. This leads to multiple iterations of the ready list. In the worst case, when all N tasks are ready, the algorithm performs N scheduling steps, in the following steps reviewing the list of size $N, N-1, \ldots, 1$. Assuming that inserting the task into the schedule costs $O(N)$ (naive insertion scheduling)[1], the whole algorithm has complexity of $O(N^3)$.

In the second case, when tasks' priorities give the order in which tasks will be assigned to processors, they must obey tasks' dependencies defined by the DAG, i.e. each task must have the priority greater than its successors:

$$\forall t_i \forall t_j \in succ(t_i) : p(t_i) > p(t_j). \quad (3)$$

The algorithm is therefore simpler (see Table 1). Firstly, tasks are ordered by decreasing priority. Then, each tasks is allocated to the processor which gives the earliest time of completion. For each processor m, task's i ready time r_m^i is computed (the time when the last communication arrives). Then task i is inserted into the schedule of the processor m after the time r_m^i (not necessarily at the end of the schedule, if there is a gap long enough to insert the task). The completion time f_i^m is the sum of task's start time s_m^i and the computation time w_m^i. This algorithm inserts each task into the schedule only once, so with the naive insertion

[1] By using balanced tree for storing scheduled tasks on each processor this cost can be easily reduced to $O(\log(N))$

Table 1: List scheduling algorithm

1. Sort $taskList$ by decreasing priority
2. for each task t_i:
   (a) $bestProc = \varnothing; eft = \inf$
   (b) for each processor p_m
      i. compute ready time $r_m^i = \max_{t \in pred(t_i)}(f_t^{alloc(t)} + c_{t,i}^{alloc(t),m})$
      ii. $s_m^i = \min_{timeT} : T \geq r_m^i \wedge$ p_m is free during $(T, T + r_m^i)$
      iii. $f_i^m = s_m^i + w_i^m$
      iv. if $(f_i^m < eft)$: $bestProc = m; eft = f_i^m$
   (c) $alloc(t_i) := bestProc$

scheduling its complexity is $O(N^2)$ (which can be further reduced to $O(N \log(N))$ with balanced tree).

4 Differential Evolution

DE is a population-based global optimization method, suitable for functions defined on totally ordered spaces, including real numbers. There were many attempts to use genetic-like approach in global optimization of such functions, including the binary-encoded GA or the Evolution Strategies. Those approaches were, however, sensitive to some phenomena seen frequently in the functions to be optimized, such as epistasis (hidden dependencies between variables which do not allow the separate optimization on each dimension), rotation (a search method often failed when the function was rotated in the search space) or binary-encoding related problems (typically used encoding is often unable to represent wide range of variables' values).

The novel approach in DE is a customized mutation operator which replaces the commonly-used reinitialization from the uniform distribution. In DE's basic version, *DE/rand/1/bin*, to the gene to be mutated it is added the difference between the values of this gene from two randomly-chosen individuals. In the version used in this article, *DE/current-to-rand/1*, a third individual is used and all of the genes are changed in order to make the search process rotation-indifferent (see Table 2). The selection operator is also simplified. Given the parent and its child, the better one is chosen.

The optimization of tasks' priorities can be viewed as the minimization of a function which assigns a makespan to a vector of priorities. Although this function probably contains some domain-specific operations (such as the second step of the list scheduling algorithm, covered in the Section 3.2), we treat it like a black box. We expect that such a function will be difficult to optimize. The number of dimensions is high, because it is equal to the number of tasks. Epistasis also occur, as the schedule is produced by comparing the values of input priorities. Therefore a robust global optimization method should be used to optimize such a function – and we expect the DE is such a solution.

Table 2: Differential Evolution algorithm (*DE/current-to-rand/1*)

$x_{j,i}$	the value of jth dimension in ith individual
$x_j^{(hi)}$	upper constraint for jth dimension
$x_j^{(lo)}$	lower constraint for jth dimension
G_{max}	maximum number of generations
K	coefficient of combination (parameter of the algorithm)
F	coefficient of mutation (parameter of the algorithm)

1. Initialize the population:
   for each individual i and each dimension j:
   $$x_{j,i} = x_j^{(lo)} + rand[0,1] \cdot (x_j^{(hi)} - x_j^{(lo)})$$
2. for $genNum = 1$ to G_{max}
   (a) for each individual i:
      i. randomly choose 3 distinct individuals r_1, r_2, r_3
      ii. mutate and recombine:
      $$\overrightarrow{x_{child}} = \overrightarrow{x_i} + K \cdot (\overrightarrow{x_{r3}} - \overrightarrow{x_i}) + F \cdot (\overrightarrow{x_{r1}} - \overrightarrow{x_{r2}})$$
      iii. select: if $f(\overrightarrow{x_{child}}) < f(\overrightarrow{x_i})$:
      replace individual i by $child$ in the next generation.

5 Differential Scheduler

Our algorithm employs the idea of differential evolution to optimize priorities of nodes, which are the input data for the second phase of the list scheduling algorithm. Differential Scheduler (DS) replaces the heuristics used in the step of priority computation in the list scheduling algorithm (Section 3.2) by a meta-heuristic. It uses internally the second step of the list scheduler in order to evaluate the proposed priorities (to compute the length of the schedule). The result of the algorithm are the optimal values for priorities of tasks. In order to obtain the schedule induced by those priorities it suffices to execute the evaluation function (i.e. the second step of the list scheduler) with the result. The whole algorithm is described in the Table 3.

The length of an individual is equal to the number N of tasks in the DAG. kth gene of ith individual specifies a single real number – $pr_i(t_k)$, the value of the *priority* for the kth task (see Fig. 2).

The search space defined in such a way is sizable. Therefore it might be valuable to start the global search

Table 3: Differential Scheduler algorithm

1. for each priority computing heuristic

   $H \in \{\text{upward ranking}, \text{downward ranking}\} \times$
   $\times \{mean, median, simplebest, best,$
   $simpleworst, worst\}$

   (a) initiate $NumCopy$ clones of an individual with priorities computed by H

   (b) if H is downward ranking: inverse priorities

   (c) normalize the individuals

2. initiate the rest of population randomly:

   (a) assign random priorities to each individual

   (b) apply repair algorithm to each individual (Table 4)

   (c) normalize each individual

3. for each individual i:
   compute schedule length by running the list scheduler with priorities $\vec{p_i}$

4. for $genNum = 1$ to G_{max}

   (a) $newPopulation = \varnothing$

   (b) for each individual i of current population pop

      i. randomly choose 3 other individuals r_1, r_2, r_3

      ii. create a child with priorities:

$$\overrightarrow{pr_{child}} = \overrightarrow{pr_i} + K \cdot (\overrightarrow{pr_{r3}} - \overrightarrow{pr_i}) + \\ + F \cdot (\overrightarrow{pr_{r1}} - \overrightarrow{pr_{r2}})$$

      iii. apply repair algorithm (Table 4)

      iv. normalize the child

      v. compute schedule length $makespan_{child}$ by running the list scheduler with priorities $\vec{p_{child}}$

      vi. if $makespan_{child} < makespan_i$
      $newPopulation \cup = child$
      else
      $newPopulation \cup = i$

   (c) $pop = newPopulation$

from a number of "acceptable" solutions, rather than from random ones (step 1a of the algorithm). Priorities computed by different heuristics outlined in the Section 3.1 (i.e. either upward ranking (Eq 1), or downward ranking (Eq 2) with one of the functions *mean*, *median*, *simple best*, *best*, *simple worst*, *worst*), seem to be natural candidates to be included in such a starting set. What is more important, "vector of priorities" is a common language of all priority-computation based list scheduling algorithms, so it is a natural base for comparing and merging their results.

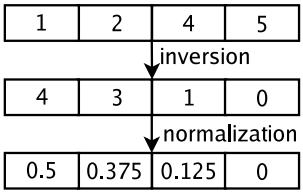


Figure 2: An example of an individual for the DAG from Fig. 1 with priorities computed by a downward ranking (top), after inverse operation (middle) and after normalization (bottom).

Of course different heuristics can assign values which differ numerically, e.g. we can easily construct an heuristic H1 which assigns priorities from the range $(0, 1)$ and an heuristic H2 from the range $(0, 1000)$. However, in such a vector, the information is stored in relations (for the example from Fig. 2, bottom: $pr(t_4) < pr(t_2)$, $pr(t_1) > pr(t_2)$) between priorities (t_4 after t_2, t_1 before t_2), and not the values themselfs. Therefore, operations such as multiplying the vector by a number (i.e. multiplying all the priorities) or adding a scalar value to the vector do not change the schedule returned by the list scheduler in the second step. Hence, after applying a heuristic, in the step 1c we *normalize* the vector by subtracting the minimum value and by dividing the vector by the sum of priorities:

$$pr(t_i) = \frac{pr(t_i) - \min_{k=1\ldots|N|} pr(t_k)}{\sum_{k=1}^{|N|} pr(t_k)} \tag{4}$$

Our list scheduler requires that the priority of each task has to be greater than priorities of all its successors – this is the easiest way to ensure that the dependencies between tasks are not violated. Priorities computed by downward heuristics (Eq. 2) are, however, inverted, in the sense that for each task, the priority of a task is less than the priority of its successors. Therefore, priorities are inversed (step 1b) before the normalization step described in the previous paragraph:

$$pr(t_i) = \left(\max_{k=1\ldots|N|} pr(t_k) \right) - pr(t_i). \tag{5}$$

The rest of the population is initialized randomly (step 2).

In order to compute the makespan (steps 3 and 4(b)v), individual's priorities are the input data to the second step of the list scheduler algorithm, described in the Section 3.2.

Population of individuals is evolved by a DE algorithm (loop in the step 4). In each generation, for each individual i, three other individuals $r1, r2, r3$ are randomly selected. Those four individuals produce a child, whose priority values are a product of parents' priorities:

$$\overrightarrow{pr_{child}} = \overrightarrow{pr_i} + K \cdot (\overrightarrow{pr_{r3}} - \overrightarrow{pr_i}) + F \cdot (\overrightarrow{pr_{r1}} - \overrightarrow{pr_{r2}}),$$

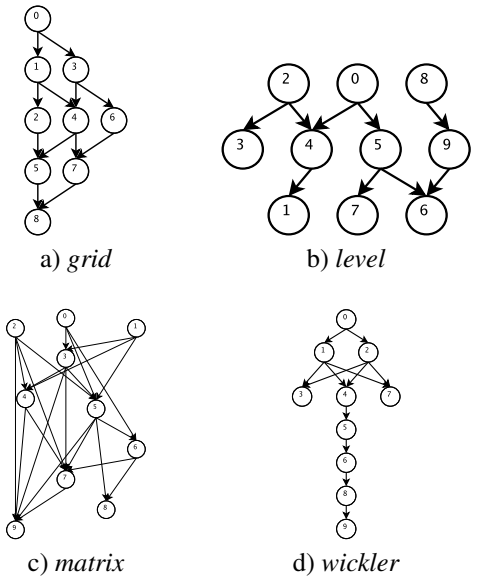
Table 4: The algorithm for repairing the priorities of tasks

$toVisit$	a FIFO queue of tasks to be visited
$uSN(t)$	number of unvisited successors for each task t
$NVPreds$	for task t: set of predecessors with priority greater than $pr(t)$
C	a number greater than 1 (we used 1.1)
ϵ	a small number (used in border cases)

1. initiate $toVisit$ with all tasks with no successors
2. for each task t: $uSN(t) = |succ(t)|$
3. while not empty($toVisit$)
   (a) task t = first task from $toVisit$
   (b) foreach $t_i \in pred(t)$: $uSN(t_i)$ $--$
   (c) foreach $t_i \in pred(t) \wedge uSN(t_i) = 0$: add t_i to $toVisit$
   (d) $minPred = \min_{t_i \in pred(t)} pr(t_i)$
   (e) $NVPreds = \{t_i : t_i \in pred(t) \wedge pr(t_i) > pr(t)\}$
   (f) if $NVPreds \neq \varnothing$ $minNVPred = \min_{t_i \in NVPreds} pr(t_i)$
   (g) if ($minPred > pr(t)$) continue;
   (h) $toAdd = pr(t) - minPred$
   (i) if ($NVPreds \neq \varnothing$) $toAdd += \frac{1}{2}(minNonViol - pr(t))$ else $toAdd = toAdd * C$
   (j) if ($toAdd < \epsilon$) $toAdd = \epsilon$.
   (k) add $toAdd$ to the priority of all the predecessors

where $\overrightarrow{pr_x}$ is a vector of priorities of the individual x (note that this is an operation on the vector of priorities). Such an operation can produce an unfeasible child, i.e. it is possible that for some tasks the condition (Eq. 3) is violated. Therefore, after the crossover the new individual is repaired (step 4(b)iii) by an algorithm described in Table 4. Without priorities' reparing, we would have to use the slower version of list scheduler. The child replaces its parent i, if its makespan is shorter than parent's.

The repair algorithm (Table 4) assigns the values for priorities which result in ordering as close as possible to the original one, but which obeys the rule (Eq. 3). The cost of this algorithm is $O(N)$. The algorithm visits, from the bottom till the top, every task t in the graph. For each t, two predecessor tasks are found: $minPred$ which has the minimum priority and $minNVPred$, which has the minimum priority from the tasks which do not violate the rule in Eq. (3). If every predecessor obeys the rule (3), their priorities do not have to be adjusted. Otherwise,



a) *grid* b) *level* c) *matrix* d) *wickler*

Figure 3: Examples of small graphs from different families

$toAdd$ contains the value which will be added to every predecessor. This value is mainly the difference between the priorities of $minPred$ and t. In order to increment it even more (so that $pr(minPred) > pr(t)$), the algorithm adds either half of the difference between $pr(minNVPred)$ and $pr(t)$, or (if all the predecessors violate the rule (3), $pr(t) - pr(minPred)$ multiplied by a constant.

6 Experimental Results

We performed an extensive simulation in order to evaluate the quality of the results obtained by our algorithm. We experimented on four families of random graphs (depicted in Figure 3), with three different CCR ratios (0.1, 1, 10), and different number of nodes (50, 100, 200, 500). There were 5 graphs generated for each combination of those parameters. The processing time of each task on each processor w_i^m were randomly chosen from the range $(1, 20)$ with the uniform distribution. The algorithms were scheduling those graphs on three different system architectures (two processor, fully connected four processors and eight processors connected in cube). In total, there were 675 different experiment settings (we have not experimented on matrix graphs with 500 nodes because of the long time of execution of the list scheduling algorithm).

We compared our algorithm with HEFT and a slightly modified genetic algorithm (GA) from [2]. For each graph, we compared the best result of the HEFT (the minimum makespan over the results returned by all the 12 possible heuristics described in the Section 3) with the result of DS and the GA by computing a percentage improvement of the makespan:

$$gain = 100\% * \frac{makespan_{HEFT} - makespan_{algorithm}}{makespan_{HEFT}}$$

We ran both the GA and the DS five times on each graph.

Table 5: Comparison of the percentage gain on the schedule length produced by the Differential Scheduler (DS) and the Genetic Algorithm (GA) with regard to the HEFT schedule length

factor		DS best [%]	DS average [%]	GA best [%]	GA average [%]
average		5.34	4.55	2.64	1.34
system	two processor	4.88	4.17	2.96	1.87
	four processor	5.52	4.65	2.89	1.39
	eight processor cube	5.63	4.83	2.09	0.76
graph family	grid	5.96	5.05	2.11	0.58
	level	6.34	5.41	2.79	1.49
	matrix	1.29	1.02	-1.36	-2.12
	wickler	6.77	5.85	6.03	4.54
graph size	50	6.18	5.38	3.46	1.60
	100	5.24	4.39	2.28	0.99
	200	5.13	4.39	2.55	1.57
	500	4.66	3.88	2.17	1.14
CCR	10	8.19	7.16	3.79	1.94
	1	4.76	3.90	2.42	1.24
	0.1	3.13	2.65	1.75	0.84

We compared both the best makespan found in 5 runs and the average makespan. As the number of experiments is considerable, Table 5 presents only aggregated results. When analyzing the impact of one of the dimensions (family, CCR, number of nodes, system) on the results, the results from other dimensions are averaged (e.g. the second row of the Table 5 presents an average over all the experiments with two processor system).

We had to modify the original GA from [2] because the list algorithm used there was optimized for homogeneous model. In each step tasks were scheduled only on processors which were free in that moment, which not necessarily results in the processor with earliest time of completion. Such a version of the GA gave results worse than the HEFT algorithm. We decided to apply the same repair algorithm we used in DS, but without modifying permanently the values of the genes (i.e. priorities are modified only for the list scheduler).

We set the population size of DS to 24 and number of generations to 50 – after some basic experiments we saw that those values are small enough to get decent execution time, yet sufficient to yield acceptable results. Those settings result in 1224 function evaluations in the course of optimization. For the initialization of the population, DS used all 12 heuristics, each heuristic produced 1 individual ($numCopy = 1$). The other 12 individuals were initiated randomly. The coefficients of combination (K) and mutation (F) were set to 0.5. We have seen though that the algorithm was not very sensitive when those values were from the range $(0.1, 0.9)$. We set the parameters of the GA as suggested by its authors, however, in order to have fair comparison with the DS, we set the same population size and the same number of generations.

The average time of execution of the DS was 10.8 seconds, considerably more than the average time of the HEFT (140 ms). It varied with the size and the family of the graph.

Table 5 summarizes the results obtained. On average, DS improves the makespan of the schedule by 4.55%, which is considerably better than the GA (1.34%). In the case of DS, the average from 5 runs was close to the best result obtained (5.34%), contrary to the GA, where the differences were much more significant – 2.64% of improvement when looking at the best result and only 1.34% when looking at the average. DS performs almost equally well on every type of parallel system considered – however a slight increase in the quality with the increased number of processors can be seen (4.17%, 4.65%, 4.83%). Because the DS algorithm does not encode the processor number, we think that either HEFT performs worse on larger systems, or the list scheduler algorithm used by DS has more "freedom" to choose the right processor. The results of the GA are worsening quickly with the increased number of processors. Both algorithms had problems with *matrix* type graphs – in the case of DS, the average improvement in that graphs (1.02%) is more than five times lower than the average improvement on the three other types of graphs (5.43%). GA's results were worse (by 2.12%) than the HEFT's. We suspect that, due to a large number of edges in those graphs, the HEFT results are close to the optimum, that there is no much place left for improvements. It is also possible that that both algorithm do not deal well with highly connected graphs. The results of the DS algorithm are worsening when increasing the number of nodes. When the the number of nodes increases, the search space and the length of the chromosome gets larger. As the number of function evaluations remains the same, one can expect such behaviour. Both algorithms achieved best results on graphs in which communication plays important role. The difference between DS's results on graphs with $CCR = 10$ (7.16%) and $CCR = 0.1$ (2.65%) is especially signficant. We suspect that this is due to the fact of underestimating the costs of communication by the HEFT. In such cases DS has "a room for improvement".

When the other dimensions are not averaged, our algorithm can improve the results of the HEFT from 0%

Table 6: Comparison of the percentage gain on the schedule length produced by the Differential Scheduler (DS) and the best result found (OPT) with regard to the HEFT result

factor	DS best [%]	DS average [%]	OPT [%]
average	7.34	6.27	8.90
graph family			
level	6.87	5.95	8.68
grid	6.32	5.27	7.43
wickler	8.83	7.59	10.60
graph size			
50	7.77	6.67	8.94
100	7.60	6.39	9.26
200	6.65	5.74	8.50
CCR			
0.1	5.20	4.38	6.73
1	4.99	4.02	6.35
10	11.82	10.40	13.63

(some *matrix* graphs) up to 12%-17% (some graphs with 50 nodes and CCR=10).

In the second set of experiments we wanted to observe how far the proposed solutions are from the ones obtained with "decent" parameter settings (which should be closer to the optimal solutions). We ran DS algorithm with population size of 100 individuals and with 200 generations at most (which resulted in 20077 function evaluations). As in the previous set of experiments we saw that the DS results are independent of the parallel system, so we decided to experiment only on four processor system. We also chosen smaller graphs in order to diminish the time required for experiments (remembering that on bigger graphs the differences between the "optimal" and the "normal" results should be bigger). Table 6 summarizes the results obtained. OPT is the best result of the DS with "decent" parameters found in 5 independent runs.

We can see that, on average, the difference between the gain obtained by the DS (6.27%) and the OPT value (8.90%) is not very big. On the other side, OPT version requires almost eight times as much computational time. Similarly to DS, OPT acheives best results on the *wickler* graph family, though the difference between those two results is the biggest. We think that this graph family has the biggest possibility of improvement. The difference in gains between the graphs with CCR=0.1 and CCR=10 also probably results from a similar phenomenum. To our surprise, the difference between the DS and the OPT results is not the biggest on the biggest programs.

7 Conclusions and Future Work

We have presented an optimization algorithm for the problem of DAG scheduling which uses the differential evolution to optimize the priorities of the individual tasks. Our algorithm starts with a number of "acceptable" solutions (results of different heuristics) and gradually improves them. The algorithm works with rather small population and a limited number of generations. From the results of extensive experiments performed we can conclude that the algorithm is not very sensitive neither to the size of the system, nor to the number of nodes in the graph and on the most of the graphs considered it is able to improve the results of the greedy heuristics on average by 4.5%. The results are close to the optimal ones, obtained with much higher cost. By using a representation close to the problem domain, together with genetic operators acting directly "on" that representation, we outperformed a GA using the same search space.

In our future work we would like to apply other meta-heuristics to the same search space. We also plan to add new heuristics for computing priorities in order to further expand the initial population.

Bibliography

[1] I. Ahmad and M. K. Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Comput.*, 22(3):395–406, 1996.

[2] M. Dhodhi and I. Ahmad. A multiprocessor scheduling scheme using problem-space genetic algorithms. In *Evolutionary Computation, 1995., IEEE International Conference on*, pages 214–219, 1995.

[3] H. El-Rewini, T. Lewis, and H. Ali, editors. *Task Scheduling in Parallel and Distributed Systems*. PTR Prentice Hall, 1994.

[4] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[5] V. Kianzad and S. Bhattacharyya. Multiprocessor clustering for embedded systems. In *Proceedings of Euro-Par 2001*, volume 2150 of *LNCS*, pages 697–701, London, UK, 2001. Springer-Verlag.

[6] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[7] K. V. Price. An introduction to differential evolution. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 81–108. McGraw-Hill, London, 1999.

[8] F. Seredynski and A. Zomaya. Sequential and parallel cellular automata-based scheduling algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 13(10):1009–1023, 2002.

[9] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[10] A. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to

multiprocessor scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 15(9):824–834, 2004.

[11] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Proceedings of Euro-Par 2003*, volume 2790 of *LNCS*, pages 189–194. Springer, 2003.