Dynamic Local Scheduling of Multiple DAGs in Distributed Heterogeneous Systems

Ondřej Votava, Peter Macejko, and Jan Janeček

Czech Technical University in Prague
{votavon1, macejp1, janecek}@fel.cvut.cz
http://cs.fel.cvut.cz

Abstract. Heterogeneous computational platform offers a great ratio between the computational power and the price of the system. Static and dynamic scheduling methods offer a good way of how to use these systems efficiently and therefore many algorithms were proposed in the literature in past years. The aim of this article is to present the dynamic (on-line) algorithm which schedules multiple DAG applications without any central node and the schedule is created only with the knowledge of node's network neigbourhood. The algorithm achieves great level of fairness for more DAGs and total computation time is close to the standard and well known competitors.

1 Introduction

Homogeneous (heterogeneous) computational platform consists of a set of identical (different) computers connected by a high speed communication network [1, 2]. Research has been done last few years on how to use these platforms efficiently [3–5]. It is believed that scheduling is a good way on how to use the computation capacity these systems offer [1, 6]. Traditional attitude is to prepare the schedule before the computation begins [7, 8]. This requires information about the network topology and parameters and also node's computational abilities. Knowing all of this information we can use the static (offline) scheduling algorithm. Finding the optimal value of *makespan* – i.e. the time of the computation in total – is claimed to be NP complete [9, 10]. Therefore research has been done and many heuristics have been found [11, 12].

Compared to static scheduling, dynamic (online) scheduling allows us to create the schedule as part of the computation process. This allows dynamic algorithms to use the feedback of the system and modify the schedule in accordance with current state of the system. Dynamic algorithms are often used for scheduling multiple applications [13–16] at the same time and therefore fairness of generated schedules is important.

The algorithm presented in this paper does not require the global knowledge of the network, it uses the information gathered from node's neighbors only. The phase of creating the schedule is fixed part of the computation cycle. The algorithm is intended to be used for scheduling more DAGs simultaneously and tries to achieve fair division of tasks for all computing nodes.

M. Nečaský, J. Pokorný, P. Moravec (Eds.): Dateso 2015, pp. 1–12, CEUR-WS.org/Vol-1343.

The structure of this article, which is the enhanced version of [17], is as follows, in the section two we describe the problem of scheduling and make a brief summary of related work. In the section three we describe the algorithm itself and in the following section we describe the testing environment and results we obtained by running several simulations. In the fifth section we conclude the results from section four, show the pros and cons of the presented algorithm and discuss the future improvements.

2 Problem definition

The application model can be described as a directed acyclic graph $AM = (\mathbf{V}, \mathbf{E}, \mathbf{B}, \mathbf{C}$ [12, 18], where:

- $\mathbf{V} = \{v_1, v_2, \ldots, v_v\}, |\mathbf{V}| = v$ is the set of tasks, task $v_i \in \mathbf{V}$ represents the piece of code that has to be executed sequentially on the same machine,
- $\mathbf{E} = \{e_1, e_2, \ldots, e_e\}, |\mathbf{E}| = e$ is the set of edges, the edge $e_j = (v_k, v_l)$ represents data dependencies, i.e. the task v_l cannot start the computation until the data from task v_k has been received, task v_k is called the parent of v_l, v_l is called the child of v_k,
- $\mathbf{B} = \{b_1, b_2, \ldots, b_v\}, |\mathbf{B}| = v$ is the set of computation costs (e.g. number of instructions), where $b_i \in \mathbf{B}$ is the computation cost for the task v_i,
- $\mathbf{C} = \{c_1, c_2, \ldots, c_e\}, |\mathbf{C}| = e$ is the set of data dependency costs, where $c_j = c_{k,l}$ is the data dependency cost (e.g. amount of data) corresponding to the edge $e_j = (v_k, v_l)$.

The task which has no parents or children is called entry or exit task respectively. If there are more than one entry/exit tasks in the graph a new virtual entry/exit task can be added to the graph. Such a task would have zero weight and would be connected by zero weight edges to the real entry/exit tasks.

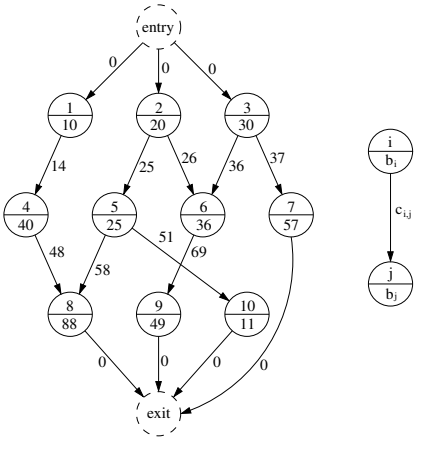

Fig. 1. Application can be described using DAG.

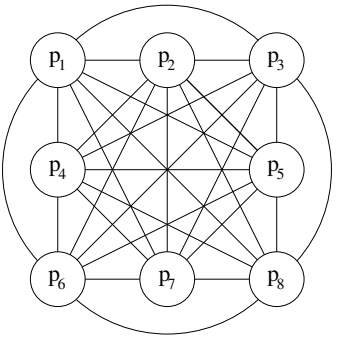

Fig. 2. The computation system can be represented as a general graph.

Since the application can be described as DAG we use terms *application*, *DAG application* or simply *DAG* as synonyms in this paper.

The computation system consist of a set of computing units all of which are connected by a high speed communication network. It can be described using a general graph $CS = (\mathbf{P}, \mathbf{Q}, \mathbb{R}, \mathbb{S})$, where:

$\mathbf{P}$ $= \{p_1, p_2, \ldots, p_p\}, |\mathbf{P}| = p$ is a set of the computers,
$\mathbf{Q}$ $= \{q_1, q_2, \ldots, q_p\}, |\mathbf{Q}| = p$ is the set of speeds of computers, where q_i is the speed of computer p_i,
$\mathbb{R}$ is a matrix describing the communication costs, the size of $\mathbb{R}$ is $p \times p$,
$\mathbb{S}$ is a matrix used to describe the communication startup costs, it is usually one dimensional, i.e. it's size is $p \times 1$.

2.1 Merging application and computation model

Since the structure of the application and the characteristics of the computation systems are known it is no problem to get the information about computation time of each application's node at any computation node. This information is stored in a $\mathbb{W}$ matrix, whose dimensions are $v \times p$, where the item at the position $[i, j]$ contains the information about the length of the computation of the task i on a computation unit j. The value of $\mathbb{W}[i, j]$ is computed by the following equation

$$\mathbb{W}[i, j] = \frac{b_i}{q_j}, \tag{1}$$

where b_i is the computation cost of task v_i and q_j is the speed of computer p_j (e.g. instructions per second).

The total communication time for a message m (corresponding data dependency for the edge (v_k, v_l)) that is send from the computer i to the computer j can be computed by this equation

$$c_m = \mathbb{S}[i] + \mathbb{R}[i][j] \cdot c_{k,l}. \tag{2}$$

2.2 Our model description

The content of the matrix $\mathbb{W}$ is dependent on the properties of computation nodes. However, the computers differ only in a certain ways. The "fast" computers are k times faster than "slow" computers. On that account the columns in the $\mathbb{W}$ are usually only multiples of one column. This information can be reduced to the constant k_p for each processor p. When the computation system is allowed to change, the matrix $\mathbb{W}$ is not useable either since it does not reflect any dynamic behaviour.

The structure, we decided to use, can be described as follows. We selected one processor to serve as a reference – p_{ref}. The computation time of one instruction on this processor lasts one time unit. The speedup of the processor p_i is then defined as

$$SU_p(i) = \frac{q_i}{q_{ref}}, \tag{3}$$

where q_i is the speed of processor p_i and q_{ref} is the speed of the reference processor.

The time duration of computation of a task v_j on the processor p_i is then computed "on the fly" by the equation

$$time_{j,i} = \frac{b_j}{SU_p(i)} \ . \tag{4}$$

Finally, the computation platform is described as a set of speedups and the communication matrices and the merging of application and computation model is being done as a part of the computation. Even the communication matrices may be reduced in our model. They contain only information about computation node's neighbours and differ for all nodes. Still, this is a problem for implementation part and does not affect the description model, as the "neighbour's" matrices are only a part of "global" communication matrices.

2.3 Related work

Task scheduling or task mapping has been in active research for a long time. Several static algorithms were introduced and dynamic algorithms were published too. Most of static scheduling algorithms are designed to work with one DAG. List scheduling algorithms are very popular and they are often used. HEFT [19] is a simple and effective algorithm used as a reference in our article. HEFT creates a list of tasks sorted by an upward rank[1] and then it assigns tasks to the processor so that the execution time of the task is minimized. Another algorithm presented in [19] is Critical Path On a Processor (CPOP). This algorithm is more complex and optimizes tasks on a critical path. By modifying list algorithms and allowing the execution of tasks more than once tasks duplication algorithms were introduces. HCPFD [2] compared to the HEFT obtains better makespan in most cases. Task duplication achieves surprisingly good results when applied to the computation model containing multi core computers [18].

The way of computing multiple DAGs is usually presented in dynamic algorithms. In [20] there was introduced a static method how to schedule multiple DAGs and the aim was not only to optimize makespan but also to achieve the fair sharing of resources for the competing DAGs. The idea of generating a new graph by appending whole DAGs to the current one is used in [21]. Compared to [20] this algorithm is dynamic, i.e. the graph is build when a new DAG arrives to the system.

Truly dynamic algorithm is described in [14]. This algorithm divides the nodes into two groups. The first one contains nodes used for computation, the second one contains scheduling nodes. Scheduling nodes are independent and the knowledge about the activity of other scheduling nodes is received through the statistics of usage of the computing nodes. The quality of such scheduling is then dependent on the quality of statistics created by computation nodes.

[1] See [19] for details

Unlike the previous one the algorithm presented in [16] is based on one central scheduling unit. The algorithm takes into account the time for scheduling and dispatching and focuses on reliability costs. Another model of completely distributed algorithm is presented in [15]. This algorithm divides nodes into groups and uses two levels of scheduling. The high level decides which group to use and low level decides which node in the group to use.

The algorithm described in [22] works a bit different way. The node works with it's neighborhood and the distribution of task of parallel application is done according to the load of the neighbours. If the load of a node is too high, the algorithm allows the task to be migrated among the network. Genetic programming technique is used to decide where to migrate the task.

The problem of task scheduling is loosely coupled with the network throughput. The description of network used in this paper is not very close to the reality and the problems connected to bottle necks or varying delay may cause problems. The behaviour of task scheduling applications running in the network, which has different parameters, is very well described in [23]. According to this article we expect there are no bottle necks in the networks.

3 Proposed algorithm

In this section we present the algorithm Dynamic Local Multiple DAG (DLMDAG). The algorithm itself, described in Algorithm 1, is a dynamic task scheduling algorithm that supports both homogeneous and heterogeneous computation platforms.

The main idea of the algorithm is based on the assumption that the communication lasts only very short time compared to the computation (at least in one order of magnitude). The computation node, which is the creator of a schedule for a certain DAG application, sends a message to all of its neighbours where it asks how long would the computation of these tasks last if they were computed by the neighbour. Then it continues computing the task and during this computation replies for the question arrive. According to the data (timestamps) received, the node makes a schedule for the set of tasks (asked in previous step), then it sends a message to it's neighbours with the information about who should compute which task and generates another question about the computation time for the next set of tasks.

The algorithm description (Algorithm 1) uses these terms. The task is called "ready" when all of its data dependencies are fulfilled. Ready tasks are stored in a $tasksReady$ priority queue. The criterion for ordering is the time computed by $computePriority$. The task that is ready and is also scheduled should be stored in a $computableTasks$ queue. Each task's representation contains one priority queue for storing the pair information about finish time and neighbour at which the finish time would be achieved. The queue is ordered by the time.

$computePriority$ method is used to make the timestamps for the tasks. It is computed when a DAG application comes to the computation node (p_k) and it

Algorithm 1 The core

```
1: neighbours[], readyTasks {priority queue of tasks ready to compute}
2: computableTasks {queue of scheduled tasks for computing}
3: if not initialized then
4:     neighbours = findNeighbours()
5:     initialized = true
6: end if
7: if received DAG then
8:     computePriority(DAG) {Priority of tasks by traversing DAG}
9:     push(findReadyTasks(DAG), readyTasks)
10: end if
11: if received DATA then
12:     correctDependecies(DATA)
13:     push(findReadyTasks(DATA.DAG), readyTasks)
14: end if
15: if received TASK then
16:     push(TASK, computableTasks)
17: end if
18: if received REQUEST then
19:     for all task ∈ REQUEST do
20:         task.time = howLong(task) {time for task + time for tasks in computableTasks}
21:     end for
22:     send(REQUEST-REPLY, owner)
23: end if
24: if received REQUEST-REPLY then
25:     for all task ∈ REQUEST-REPLY do
26:         push((task.time, sender), task.orderingQueue)
27:     end for
28: end if
29: loop {The main loop of algorithm}
30:     schedule = createSchedule(tasksReady) {Creates schedule and removes tasks from queue}
31:     for all (task, proc) ∈ schedule do
32:         send(task, proc)
33:     end for
34:     for all n ∈ neighbours do
35:         send(REQUEST, n) {tasks from tasksReady}
36:     end for
37:     TASK = pop(computableTasks)
38:     compute(TASK)
39:     send(DATA, TASK.owner) {Nothing is send if local task}
40: end loop
```

is generated according to this equation

$$priority(v_j) = time_{j,k} + \max_{\forall i \in par_{v_j}} priority(i), \tag{5}$$

where par_{v_j} is the set of parents of node v_j and $priority(v_0) = time_{0,k}$.

The final scheduling is based on the priority queue *task.orderingQueue*. The scheduling step described in Algorithm 2 is close to HEFT [19]. One big difference is that our algorithm uses the reduced list of tasks[2] and is forced to use all neighbours[3] even if it would be slower than computing at local site.

The algorithm is called local. It is because it uses only information about the node's local neighborhood. Each node creates a set of neighbours in the initialization stage of the algorithm. Therefore there are no matrices $\mathbb{R}$ and $\mathbb{S}$ or there are these matrices but they are different for each computational node.

[2] Only ready tasks are scheduled

[3] If there are not enough ready tasks then not all neighbours are used.

Algorithm 2 Scheduling phase

```
schedule {empty set for pairs}
num = min(|tasksReady|, |neighbours|)
for i = 0; i < num; i + + do
    task = pop(tasksReady)
    proc = pop(task.orderingQueue)
    push((task, proc),schedule)
    removeFrom(proc, tasksReady) {Once neighbour used it cannot be scheduled again}
end for
return schedule
```

The size of matrix $\mathbb{R}$ for the computational node p_i is $\mathbb{R}_{p_i} = (s_i \times s_i)$ where $s = |neighbours_i|$ is the amount of neighbours of the node p_i.

3.1 Time complexity

The time complexity of the algorithm can be divided into two parts. The computation part is connected to the sorting and scheduling phase of the algorithm and the communication part is connected to the necessity of exchanging messages for the scheduling phase. The DAG consists of v tasks and the computation node has s neighbours. One question message is sent about every task to all of the neighbours. Question contains information from one to s tasks and the precise number is dependent on the structure of the DAG. The node which receives the question message always sends a reply to it. As the node finishes the scheduling phase of the algorithm another message with a schedule is sent to every neighbour who is involved in the schedule. The last message (schedule information) can be put together with the question's one and there is from $\mathbf{3\,v/s}$ to $\mathbf{3\,v}$ messages sent in total.

Computation part is based on the sorting operations of the algorithm. There are two types of priority queues being used all of which are based on the heap. The first one is the *tasksReady*. Every task from a DAG is put once in this queue and the time complexity is $O(v \log v)$. The second priority queue (*task.orderingQueue*) stores one piece of information for every neighbour. The queue is used for every task and for every neighbour and the time complexity obtained by this queue is $O(v\,s \log s)$ and therefore the time complexity of the computational part of the algorithm is $\mathbf{O(v \log v + v\,s \log s)}$.

4 Performance and comparison

The algorithm was implemented in a simulation environment [24] and it was executed several times for different scenarios. Makespan, unfairness and average utilization of computing nodes were measured.

Makespan is the total computation time of the application, it is defined as

$$makespan(DAG) = finishTime(v_l) - startTime(v_s), \tag{6}$$

where $finishTime(v_l)$ is the time when the last task of DAG was computed and $startTime(v_s)$ is the time when the first task of DAG began the computation.

Since several DAG applications compete for the shared resources the execution time for each DAG is longer compared to the execution time when there was the only one application in the system. The slowdown of the DAG represents ratio of the execution time when only one DAG was in system and when there were more in the system. It is described as

$$Slowdown(DAG) = T_{shared}(DAG)/T_{single}(DAG), \tag{7}$$

where T_{shared} is the execution time when more than one DAG was scheduled and T_{single} is the execution time when there was only this DAG scheduled. The schedule is fair when all of the DAGs achieve almost the same slowdown[20] and the schedule is unfair when there are big differences in the slowdown of DAGs. The unfairness for the schedule S for a set of n DAGs $A = \{DAG_1, DAG_2, ..., DAG_n\}$ is defined

$$Unfairness(S) = \sum_{\forall d \in A} |Slowdown(d) - AvgSlowdown|, \tag{8}$$

where average slowdown is defined as

$$AvgSlowdown = \frac{1}{n} \sum_{\forall d \in A} Slowdown(d) \tag{9}$$

The utilization of a computation unit p_j for the schedule S is computed by this equation:

$$Util_S(p_j) = \sum_{\forall t \in tasks_S} makespan(t)/totalTime_j, \tag{10}$$

where $tasks_S$ is a set of tasks which were computed on a p_j in the schedule S and $totalTime_j$ is the total time of the simulation, which is the time when the last task of all DAGs has finished.

Average utilization for the whole set of processors $\mathbf{P}$ and for the schedule S is then defined as

$$AvgUtil_S(\mathbf{P}) = \frac{1}{p} \sum_{i=1}^{p} Util_S(p_i). \tag{11}$$

4.1 Testing environment

Three computation platforms containing 5, 10 and 20 computers were created. A full mesh with different communication speed for several lines was chosen as a connection network – this created a network without bottle necks and allowed the algorithm obtain minimal makespan time [23].

Nodes were divided into groups of 5 and the group used a gigabit connection with a delay of 2 ms. In the network with ten nodes the groups were connected by 100 MBit lines and in the network with 20 nodes the groups were connected as follows:

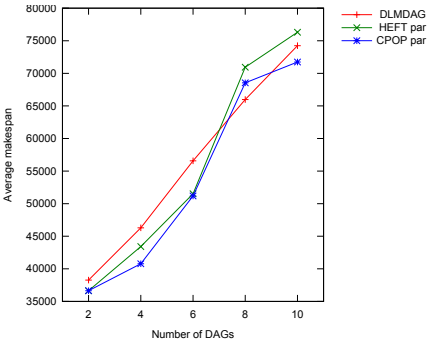

Fig. 3. Makespan for different number of DAGs running concurrently (all platforms)

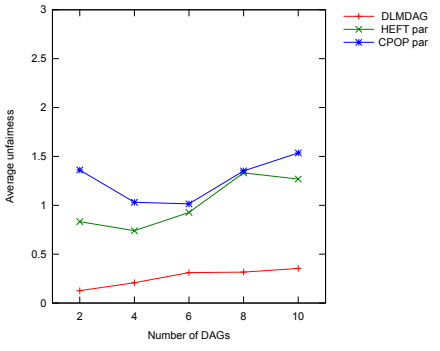

Fig. 4. Unfairness for different concurrently running DAGs (all platforms)

- 4 groups of 5 nodes intraconnected by gigabit,
- 2 groups connected by 100 MBit,
- 3^{rd} and 4^{th} group connected by 10 MBit with others.

Sets of 2, 4, 6, 8 and 10 applications were generated using the method described in [19]. The application contained 25 tasks with different computation and data dependency costs. The schedule for each of the set was generated by simulation[4] of DLMDAG algorithm and by static algorithms HEFT and CPOP.

We used two methods of connecting several DAGs into one for the static algorithms, the first one is sequence execution of DAGs in a row, the second one is to generate virtual start and end nodes and connect DAGs to these nodes with a zero weighted edges. DAGs were ordered in the sequential execution test by the rule the shorter the makespan of DAG is the sooner it is executed. In total there were 100 sets of 2 DAGs, 100 sets of 4 DAGs etc. and the results we obtained we averaged. For the DLMDAG all DAGs arrived to the system at time 0 and on the one node.

4.2 Results

Results of sequential execution of DAGs for HEFT and CPOP achieved much longer makespans and therefore were not included into graphs. HEFT par and CPOP par mean that connection of DAGs was created using virtual start and end tasks.

The makespan achieved by DLMDAG is very close to the HEFT and CPOP (fig. 3). The differences after averaging were just units of percents. The special case was the architecture of five computers (fig. 8), in this case DLMDAG outperforms the others. When there were 10 or 20 computers in the system (fig. 6), DLMDAG achieved slightly worse results. Since HEFT and CPOP use the whole

[4] Simulation tool OMNeT++[24] was used

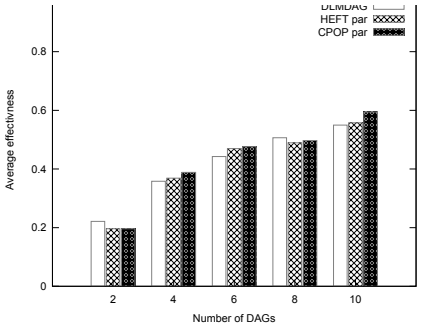

Fig. 5. Utilization for different number of DAGs running concurrently (all platforms)

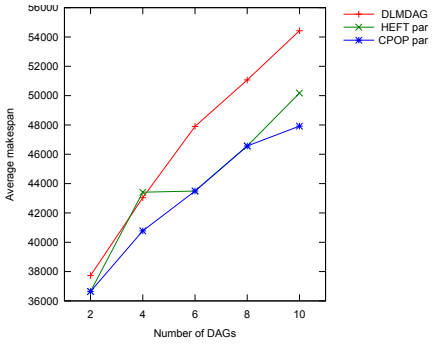

Fig. 6. Makespan for different numbers of applications (20 PC platform)

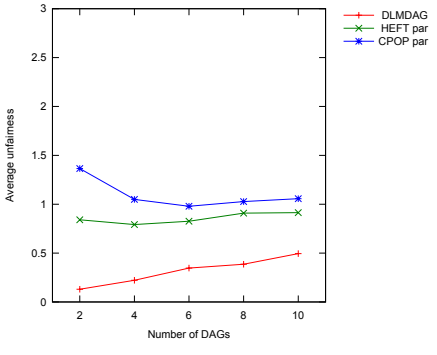

Fig. 7. Unfairness for different numbers of applications (20 PC platform)

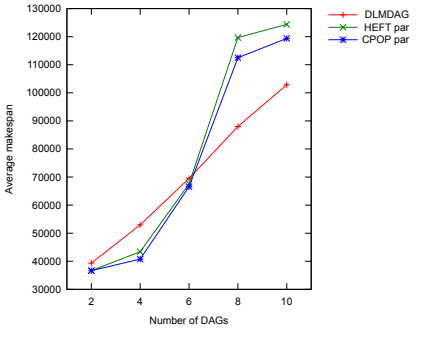

Fig. 8. Makespan for diferent numbers of applications (5 PC platform)

structure of applications DLMDAG works only with ready tasks and therefore it may be unable to use the platform with more devices so efficiently. These results are then dependent on the structure of the applications that were scheduled. The more parallel application is, the better results DLMDAG obtains.

The unfairness (figures 4, 7) for DLMDAG is at the very low level and the growth of it is slow. The unfairness level obtained by HEFT and CPOP in comparison with DLMDAG is worse.

The utilization of nodes (figure 5) corresponds to the makespan achieved by the algorithms. Growing the amount of DAGs in the system the utilization of nodes increases for all algorithms. As mentioned earlier, DLMDAG achieves high level of parallelization and therefore the average utilization of all nodes is also increasing.

5 Conclusion

The algorithm presented in this article is dynamic, it does not use any central point for scheduling neither it requires the information about the whole network.

DLMDAG is based on the local knowledge of the network – only neighbours create the schedule – and the schedule is created using several messages by which the computation times are gathered on the scheduling node. The simulations of the algorithm were executed and results obtained were compared to the traditional offline scheduling algorithms.

DLMDAG is able to use the computation resources in a better way than compared algorithms when there are more tasks in the system than computation units. As the number of computation nodes increases the result DLMDAG achieves become worse than competitor's.

Future work There are several possibilities to improve the proposed algorithm. Initially the computation systems do change. The algorithm should be able to modify the schedules to reflect the network changes. Subsequently the current algorithm is fixed to the scheduling node and it's neighbours and this may cause performance problems, the algorithm could be able to move the application to some other node with different neighbours.

References

1. D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson and L. Rudolph, eds.), vol. 1291 of *Lecture Notes in Computer Science*, pp. 1–34, Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63574-2_14.
2. T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems," *Parallel Computing*, vol. 31, no. 7, pp. 653 – 670, 2005. Heterogeneous Computing.
3. H. Kikuchi, R. Kalia, A. Nakano, P. Vashishta, H. Iyetomi, S. Ogata, T. Kouno, F. Shimojo, K. Tsuruta, and S. Saini, "Collaborative simulation grid: Multiscale quantum-mechanical/classical atomistic simulations on distributed pc clusters in the us and japan," in *Supercomputing, ACM/IEEE 2002 Conference*, p. 63
4. D. Kehagias, M. Grivas, G. Pantziou, and M. Apostoli, "A wildly dynamic grid-like cluster utilizing idle time of common pc," in *Telecommunications in Modern Satellite, Cable and Broadcasting Services, 2007. TELSIKS 2007. 8th International Conference on*, pp. 36 –39, sept. 2007.
5. A. Wakatani, "Parallel vq compression using pnn algorithm for pc grid system," *Telecommunication Systems*, vol. 37, pp. 127–135, 2008.
6. M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," in *In Encyclopedia of Electrical and Electronics Engineering*, pp. 679–690, John Wiley, 1999.
7. Y. kwong Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in *In Proc. IPPS/SPDP*, pp. 531–537, 1998.
8. J. Liou and M. Palis, "A comparison of general approaches to multiprocessor scheduling," *Parallel Processing Symposium, International*, vol. 0, p. 152, 1997.
9. J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384 – 393, 1975.
10. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

11. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810 – 837, 2001.
12. H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.
13. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," *Heterogeneous Computing Workshop*, vol. 0, p. 30, 1999.
14. M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment," *Heterogeneous Computing Workshop*, vol. 0, p. 70, 1998.
15. M. A. Iverson and F. Özgüner, "Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment," *Distributed Systems Engineering*, vol. 6, no. 3, p. 112, 1999.
16. X. Qin and H. Jiang, "Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems," *Parallel Processing, International Conference on*, vol. 0, p. 0113, 2001.
17. O. Votava, P. Macejko, J. Kubr, and J. Janeček, "Dynamic Local Scheduling of Multiple DAGs in a Distributed Heterogeneous Systems," in *Proceedings of the 2011 International Conference on Telecommunication Systems Management*, (Dallas, TX), pp. 171–178, American Telecommunications Systems Management Association Inc., 2011.
18. J. Janeček, P. Macejko, and T. M. G. Hagras, "Task scheduling for clustered heterogeneous systems," in *IASTED International Conference - Parallel and Distributed Computing and Networks (PDCN 2009)* (M. Hamza, ed.), pp. 115–120, February 2009. ISBN: 978-0-88986-783-3, ISBN (CD): 978-0-88986-784-0.
19. H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pp. 3 –14, 1999.
20. H. Zhao and R. Sakellariou, "Scheduling multiple dags onto heterogeneous systems," *Parallel and Distributed Processing Symposium, International*, 2006.
21. J. Barbosa and B. Moreira, "Dynamic job scheduling on heterogeneous clusters," in *Parallel and Distributed Computing, 2009. ISPDC '09. Eighth International Symposium on*, pp. 3 –10, 302009-july4 2009.
22. R. de Mello, J. Andrade Filho, L. Senger, and L. Yang, "Grid job scheduling using route with genetic algorithm support," *Telecommunication Systems*, vol. 38, pp. 147–160, 2008. 10.1007/s11235-008-9101-5.
23. Y. Kitatsuji, K. Yamazaki, H. Koide, M. Tsuru, and Y. Oie, "Influence of network characteristics on application performance in a grid environment," *Telecommunication Systems*, vol. 30, pp. 99–121, 2005. 10.1007/s11235-005-4320-5.
24. A. Varga *et al.*, "The omnet++ discrete event simulation system," in *Proceedings of the European simulation multiconference (ESM'2001)*, vol. 9, p. 65, sn, 2001.