Scheduling Multiple DAGs onto Heterogeneous Systems

Henan Zhao and Rizos Sakellariou
School of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
{hzhao,rizos}@cs.man.ac.uk

Abstract

The problem of scheduling a single DAG onto heterogeneous systems has been studied extensively. In this paper, we focus on the problem of scheduling more than one DAG at the same time onto a set of heterogeneous resources. The aim is not only to optimize the overall makespan, but also to achieve fairness, defined on the basis of the slowdown that each DAG would experience as a result of competing for resources with other DAGs. Two policies particularly focussing to deliver fairness are presented and evaluated along with another four policies that can be used to schedule multiple DAGs.

1 Introduction

The problem of scheduling a directed acyclic graph (DAG) onto a set of heterogeneous machines has been well studied and a number of heuristics have been proposed in the literature [12, 15, 1, 6, 9, 13, 19]. Interest in the topic has increased in recent years, partly as a result of the emergence of the so-called workflow applications as an important use case in the context of Grid Computing; some of those workflow applications can be represented by a DAG [2, 10, 20]. Grid computing provides an initial motivation for this work too. Virtually all existing work on DAG scheduling deals with the problem of scheduling a single DAG. It is reasonable to envisage a scenario where more than one DAG need to be scheduled onto resources at the same time. The only relevant work, which considers multiple DAGs, focuses on environment-related aspects but not on the appropriateness of different scheduling policies [5, 7]. This problem of scheduling multiple DAGs onto heterogeneous resources is addressed in this paper.

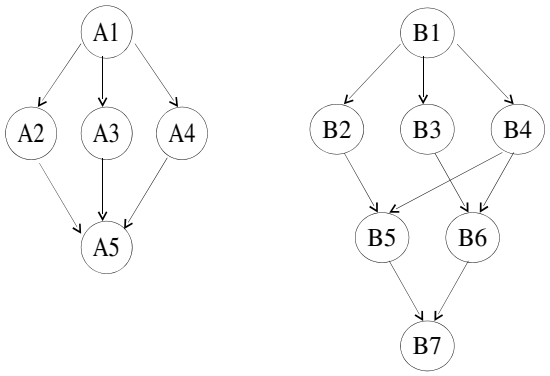
When scheduling multiple DAGs, those DAGs compete for the same resources. In this situation, although one objective would still be the minimization of the overall makespan (that is, start time of the first task of the first DAG in the set of multiple DAGs to finish time of the last task of the last DAG in the set of multiple DAGs), another objective could be to achieve a certain level of Quality of Service for the given DAGs. Such a level of Quality of Service could be based on priorities (for instance, one requirement could be to complete the execution of one DAG as soon as possible) or other constraints. In this paper, we focus on *fairness*, which is defined on the basis of the slowdown that each DAG would experience (as a result of sharing the resources with other DAGs as opposed to having the resources on its own); to achieve fairness, the aim is to make this slowdown equal for all DAGs, yet without affecting the overall makespan of the multiple DAG schedule. The paper describes two policies to schedule multiple DAGs, especially targeting at fairness. These are evaluated along with four other policies to schedule multiple DAGs (without a special consideration of fairness).

The remainder of this paper is organized as follows. Section 2 considers some basic approaches for scheduling multiple DAGs. Section 3 presents two scheduling policies which aim to deliver fairness in the schedule generated. Section 4 evaluates the methods proposed and Section 5 concludes the paper.

2 An Initial Approach

The model we use to represent a DAG, and its parameters (e.g., estimated execution time of tasks and communication costs) has been widely used in other heterogeneous computing scheduling studies [15, 18, 12]. A DAG consists of nodes and edges, where nodes (or tasks) represent computation and edges represent precedence constraints between nodes. Each DAG has a single entry node and a single exit node. There is also a set of machines (resources) on which nodes can execute (the execution time may be different on each machine) and which may need a varying amount of time to transmit data. A machine can execute only

1-4244-0054-6/06/$20.00



(a) Example DAG A (b) Example DAG B

Figure 1. Two example DAGs.

one task at a time, and a task cannot start execution until all data from its parent nodes is available. The estimated execution time of each task on each machine is given. Similarly, the amount of data that needs to be communicated between tasks is also given; along with an estimate for the communication cost between different machines, those two values (amount of data and communication cost between two machines) provide the estimated data communication cost between two tasks that have a direct precedence constraint (that is, they are linked with an edge in a DAG) and they are running on specific (different) resources. Throughout the paper, we always assume that only one task at a time can use a resource.

An obvious solution to scheduling multiple DAGs is to schedule the DAGs one after the other using any *single-DAG* scheduling algorithm [18, 15]; that is, one DAG starts execution after another finishes. The potential problem with this approach is that it may leave the resources idle for some period of time (depending on the structure of the DAG), thus resulting in a long overall makespan. A better solution is to start scheduling the tasks of each of the DAGs at the earliest possible time; thus, any idle slots in the schedule of the DAGs already scheduled might be filled by the DAGs that remain to be considered. In this situation, the order that the DAGs are scheduled may make a difference (for instance, one could start with the DAG that has the shortest makespan when running on the resources on its own, or the DAG that has the longest makespan when running on the resources on its own).

An alternative approach to the above approaches for scheduling the multiple DAGs is to schedule all the

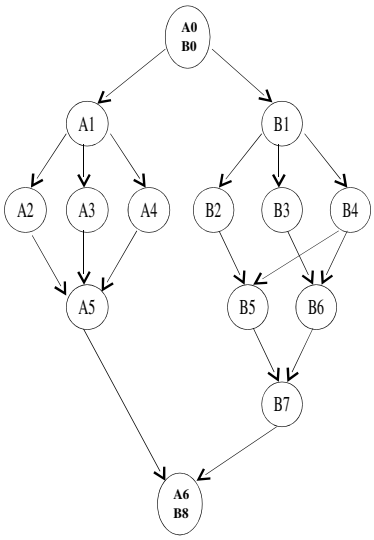


Figure 2. Composition approach 1: common entry and common exit node technique.

DAGs at the same time. There are many ways to do this; below, we focus on four approaches, which are based on merging all the DAGs into a single, composite DAG, where any single-DAG scheduling algorithm can be applied. Each of these approaches, by retaining some information from the original DAGs, can place extra constraints to the single-DAG scheduling. In order to illustrate the four approaches, consider the example DAGs in Figure 1(a) and Figure 1(b).

C1: Common Entry and Common Exit Node: This approach creates the composite graph by making the entry nodes of all the DAGs immediate successors of a new entry node, and all the exit nodes of the DAGs immediate ancestors of a new exit node, as shown in Figure 2. These two extra nodes (i.e., new entry and new exit node) have no computation and no communication between them and other nodes.

C2: Level-Based Ordering: This approach creates a composite graph in the same way as before; however, the new graph is grouped into levels as shown in the example in Figure 3. Scheduling takes place in levels; clearly, within each level, there are only independent tasks to be scheduled; these may be scheduled using any algorithm for scheduling independent tasks (see [3] for a comparison of several heuristics, or, BMCT, the heuristic suggested in [15]).

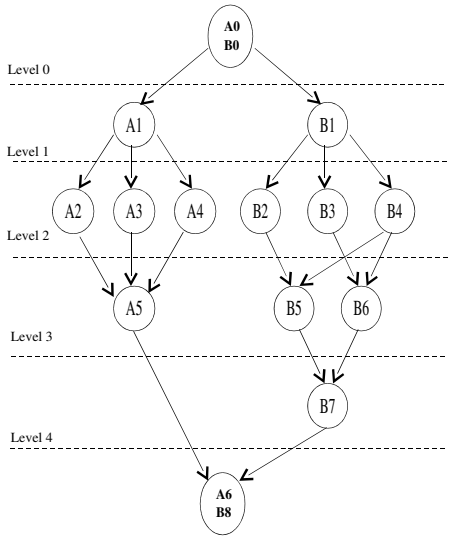


Figure 3. Composition approach 2: level-based ordering.

C3: Alternating DAGs: This approach creates a composite DAG in the same way as above; however, scheduling considers tasks from each DAG in a round-robin fashion. Thus, if at a certain point in time, the task being scheduled belongs to one DAG, tasks of the same DAG will not be considered for scheduling until a task from each of the other DAGs has been considered. To illustrate this, consider Figure 2; a possible order of scheduling might be A1, B1, A2, B2, A3, B3, etc.

C4: Ranking-Based Composition: The idea behind this approach is to take into account the structure of each DAG and the execution time of every task when linking the exit nodes. Thus, the exit node of a short DAG (that is, a DAG whose makespan when scheduled on its own is short) would be linked to that task of a long DAG, whose longest path from its entry node is about the same to the longest path of the short DAG. Longest paths can be found using upward ranking. Thus, the upward rank, $r_u(i)$, of a task i is recursively defined by

$$r_u(i) = f(w_i^0, ..., w_i^m, ..., w_i^{M-1}) + \max_{\forall j \in S_i}(f(c_{ij}^{00}, ..., c_{ij}^{mm'}, ..., c_{ij}^{M-1,M-1}) + r_u(j)),$$

where w_i^m is the (estimated) computation cost of task i on machine m, $0 \leq m < M$ (M is the total number of machines available), the function f returns the average value of its parameters, S_i is the set of the immediate successors of task i, and $c_{ij}^{mm'}$ is the communication cost between nodes i and j when i is executed by machine m and j by machine m', $0 \leq m, m' < M$ (see [21] for a more detailed specification). To illustrate this, consider the two DAGs in Figure 1, and assume that the rank of each task is as shown in Table 1. The longest path of the first DAG is 50; it can be noted that node B2 of the second DAG has a difference in terms of its rank and the rank of the entry node of the second DAG of 48. Thus, the exit node of DAG A would be linked to node B2 by adding an edge of zero

DAG	Rank			
DAG A	A1	50	A4	20
	A2	42	A5	6
	A3	36		
DAG B	B1	200	B5	45
	B2	152	B6	63
	B3	122	B7	13
	B4	140		

Table 1. The rank of each task of the two DAGs in Figure 1 using upward ranking [21].

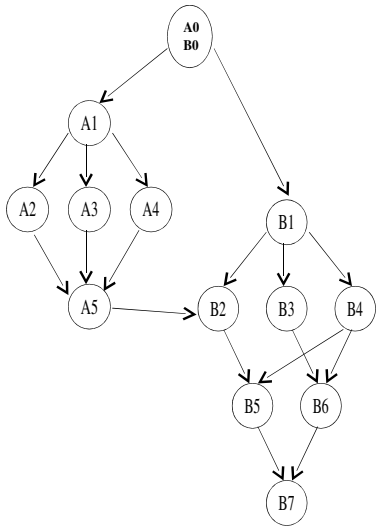


Figure 4. Composition approach 4: ranking-based composition technique.

	C1	C2	C3	C4	OnebyOne	MinMax	MaxMin
Hyb.BMCT	928.38	942.27	949.93	968.74	1420.65	1354.50	1285.74
HEFT	1059.15	1071.16	1088.85	1079.20	1689.92	1615.58	1533.26

Table 2. Average makespan, over 100 runs, of 7 different methods for multiple DAG scheduling with 2-10 randomly generated DAGs, each containing 10 to 50 tasks, running on 3 to 8 machines.

communication cost as shown in Figure 4.

For an initial evaluation of the makespan of the approaches above we carried out a small experiment using our custom-made simulator for DAG scheduling [15]. We repeated 100 times an experiment where 2 to 10 randomly generated DAGs, each containing 10 to 50 tasks, had to be scheduled at the same time. Random graphs are generated using the procedure also explained in [21, 15]. Thus, to generate a DAG with a number of nodes, we first generate a single entry and exit node; all other nodes are divided into levels, with each level having at least two nodes. Levels are created progressively; the number of nodes at each level is randomly selected up to half the number of the remaining to be generated nodes. Care is taken so that each node at a given level is connected to at least one node of the successor level and *vice versa*. It is recognized that random graphs may not always be representative, however, the diversity they provide is sufficient for this experiment.

We evaluated seven approaches: the four approaches presented above to generate a composite DAG (denoted by C1, C2, C3, C4), the naive approach of scheduling DAGs one after the other ('OnebyOne'), and two approaches for scheduling DAGs in order, but starting all of them from the earliest possible time depending on whether we start with the shortest DAG and proceed with the next DAGs in ascending order of their makespan ('MinMax') or the longest DAG ('MaxMin'). We use two heuristics for scheduling a single DAG: Hybrid.BMCT [15] and HEFT [18]. Briefly, both heuristics are based on traditional list scheduling where some prioritization of the nodes of the DAG takes place first (for example, ranking the nodes using upward ranking as above); then, following their priority order (that is, the rank of each node), each node is scheduled on the machine that gives the earliest finish time. The additional characteristic of Hybrid.BMCT is that it relaxes the priority order by considering independent tasks with adjacent priority as a group (see [15] for more details). The average makespan, from 100 runs, is shown in Table 2. It is worth noticing that there is a significant difference between the methods based on a composite DAG (C1, C2, C3, C4) and the other methods; as a result, the latter will be omitted from any further evaluation. Another interesting remark is that the schedules generated by Hybrid.BMCT lead to a significantly better makespan than the schedules generated by HEFT.

3 Dealing with Fairness

3.1 Metrics

In the context of scheduling, fairness has been used in various ways [8, 11, 14, 17]. The definition we adopted in this paper defines fairness on the basis of the slowdown each DAG would experience as a result of sharing the resources with other DAGs (as opposed to the makespan when the DAG is having the resources on its own). Fairness implies that each DAG experiences the same (or similar) slowdown. A similar definition is used in the context of resource sharing in simultaneous multithreaded architectures (SMT) where threads compete for the same resources [4].

Thus, to define the slowdown, consider a DAG a in a given set of DAGs A, a set of resources R and a schedule which allocates A to R. The slowdown value of the DAG a, $Slowdown(a)$, in this schedule is defined as

$$Slowdown(a) = M_{own}(a)/M_{multi}(a),$$

where M_{own} is the makespan of the DAG when it has the available resources on its own (later, some times referred to as *original* makespan), and M_{multi} is the makespan of the same DAG when scheduled onto resources along with all the other DAGs in the set A. The slowdown definition can also be applied to each individual task as the ratio between the finish time of a given task in the schedule where the DAG can use the resources on its own and the finish time of the same task in the schedule where the DAG shares the resources with other DAGs (assuming that in both cases the start time of the entry node in the schedule is the same). It is expected that slowdown values will be between 0 and 1 with values closer to 1 indicating a small slowdown.

Input: a set of DAGs A, a set of resources R, and a
selected list scheduling algorithm alg

Fairness policy:
(1) Run each DAG alone on R with algorithm alg. Store
the schedule for each DAG in set S_{own}.
(2) Mark each DAG as unexecuted U, and mark every task in
each DAG as unexecuted. Set the slowdown value of each
DAG as 0, and sort DAGs in descending order of
their total makespan in S_{own}.
(3) $S_{multi} \leftarrow$ stores the multiple DAG schedule
While there are unexecuted DAGs do
$a \leftarrow$ first DAG in unexecuted set U
$t \leftarrow$ first (ready) task that has not been executed on a
$m \leftarrow$ the allocated machine for t with algorithm alg
if (t is the last task of a)
remove a from U
else
$ft_{multi}(t) \leftarrow$ the finish time of task t in the multiple DAG schedule
$ft_{own}(t) \leftarrow$ the finish time of task t in S_{own}
$sd(a) \leftarrow ft_{own}(t)/ft_{multi}(t)$
Sort U in ascending order of the slowdown value of each DAG
if (two DAGs have the same slowdown value)
compare the remaining makespan for the unexecuted tasks, the one
with the highest value is scheduled before the other
endwhile

Figure 5. A Fairness Policy based on Finish Time

As mentioned above, fairness indicates that each DAG experiences the same (or similar) slowdown. Consequently, *unfairness* indicates that different DAGs experience a large variation in their individual slowdown. Thus, unfairness can be defined on the basis of the absolute value of the difference between the slowdown of each DAG and the average slowdown of all the DAGs. Hence, the unfairness of a schedule S of multiple DAGs is given by

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

where A is a set of given DAGs, and $AvgSlowdown$ is the average slowdown value for all DAGs in A given by

$$AvgSlowdown = \frac{1}{|A|} \sum_{\forall a \in A} Slowdown(a),$$

where $|A|$ denotes the cardinality of set A. A low value for unfairness may indicate that the slowdown difference between DAGs is small, i.e., the schedule is reasonably fair to each DAG.

The next section proposes two fairness policies that can be applied to generate a fair schedule for given DAGs.

3.2 Scheduling for Fairness

In this section we are going to describe two policies which aim to schedule the multiple DAGs optimizing both their makespan and fairness (that is, to minimize unfairness as defined above). The key idea is to evaluate, after scheduling a task, the slowdown value of each DAG against other DAGs and make a decision on which DAG should be considered next (that is, from which DAG a task will be picked to schedule) on the basis of which DAG shows the smallest slowdown value at this point in time. This selection policy can be applied to select the next DAG to be considered by any single DAG list scheduling algorithm.

As the tasks of given DAGs are allocated, the scheduler maintains a list that keeps track of the current slowdown value for each DAG, and sorts them in an ascending order of their value. The next DAG to be considered will be the DAG which currently has the

Input: a set of DAGs A, a set of resources R, and a
selected list scheduling algorithm alg

Fairness policy:
(1) Run each DAG alone on R with algorithm alg. Store the
finish time for each task of the DAG and the schedule, S_{orig}.
(2) Mark each DAG as unexecuted U, and mark every task in
each DAG as unexecuted. Set the slowdown value of each
DAG as 0, and sort DAGs in descending order of
their total makespan S_{orig} assigning to set SD.
(3) While there are unexecuted DAGs do
    $a \leftarrow$ first DAG in unexecuted set U
    $t \leftarrow$ first (ready) task that has not been executed on a
    $m \leftarrow$ the allocated machine for t with algorithm alg
    if (t is the last task of a)
        remove a from U
    else
        $ct \leftarrow$ the current time, i.e. finish time of t
        for all DAGs in U
            $app \leftarrow$ select a DAG from U
            $task \leftarrow$ the task which is running at time ct from DAG app
            $ct_{own}(task) \leftarrow$ schedule length including the proportion of $task$ that has been executed
            $sd(app) \leftarrow ct_{own}(task)/ct$
            store $sd(app)$ into SD
        endfor
        sort U in ascending order of the slowdown value in SD for all DAGs
        if (two DAGs have the same slowdown value)
            compare the remaining makespan for the unexecuted tasks, the one
            with the highest value is scheduled before the other
    endwhile

Figure 6. A Fairness Policy based on Current Time.

smallest slowdown value. There can be different ways to calculate the slowdown value on-the-fly, that is, after each task is scheduled; two policies are proposed here. In both policies, the component $M_{multi}(a)$ in computing the slowdown consists of the difference between the finish time of the last task that has been scheduled in the schedule with the multiple DAGs and the start time of the entry node; similarly, the component $M_{own}(a)$ considers only the schedule length of the corresponding tasks (and not the whole makespan), that is, only those tasks that have already been scheduled in the schedule with the multiple DAGs. The difference in the two policies is that the first policy, later denoted by F1, calculates the slowdown value of a DAG *only* at the time the last task that was scheduled for this DAG finished (this slowdown value will be calculated with respect to the finish time of that task in the schedule of this DAG on its own and the task's finish time in the schedule which is constructed along with other DAGs). In contrast, the second policy re-calculates the slowdown value of *every DAG* at the time the last task (of any DAG) in the schedule of the multiple DAGs finished. This task may, of course, belong to any of the original DAGs; at the same time, tasks that belong to the other DAGs may be half way through their execution. Then, in order to calculate the slowdown of each DAG, the corresponding proportion of the task that can be considered as completed at this point in time will be taken into account. The motivation for the second policy is that it would lower the slowdown of those DAGs, which did not have a task considered for scheduling for some time. The two policies are shown in Figure 5 and Figure 6, respectively.

3.3 An Example

To illustrate the two policies, we consider the two DAGs in Figure 1; suppose that we have obtained a schedule for each DAG (when scheduled on its own) with the HEFT algorithm. The multiple DAG schedule needs to make a decision at time t (see Figure 7). By that time, tasks A1, A2, A3 and A4 from the first DAG have been scheduled as well as tasks B1, and B2 from the second DAG. At time t also (which corresponds to time 53, starting from 0, the start time of tasks A1 and B1), both tasks A4 and A3 terminate. Using the first policy for fairness above, the slowdown value of the first DAG at time t is $45/53 \simeq 0.849$, the value of 45 corresponding to the finish time of task A4 in the original schedule. With respect to the second DAG, the slowdown value would be the finish time of the last scheduled task, which is task B1; this would give a slowdown value of $31/31 = 1$. It is noted here that this approach would create large variations in the slowdown values as the schedule is being generated and make them less representative of the real slowdown. In this particular example, as long as the slowdown value of the second DAG is 1, a task from the second DAG would be scheduled only if there is an empty resource and there is no task available (for instance, due to precedence constraints to be taken into account) from the first DAG (as it is the case when task B2 is scheduled).

Using the second policy for fairness above, the slowdown value of the first DAG would remain the same (since the last task of this DAG finishes at exactly the time being considered, t), however, the slowdown value of the second DAG would be $37/53 \simeq 0.698$. This would be calculated by considering the current time, t, for the denominator and, with respect to the original schedule, the time corresponding to task B2 (as a percentage that has been executed), which is being executed at time t. Given that B2 started at time 47 and its predicted running time on machine M1 is 40, the percentage of its execution that has been completed is 15%. In the original schedule, task B2 would start at time 31, hence, the value of $37 = 31 + 15\% \times 40$ in the numerator.

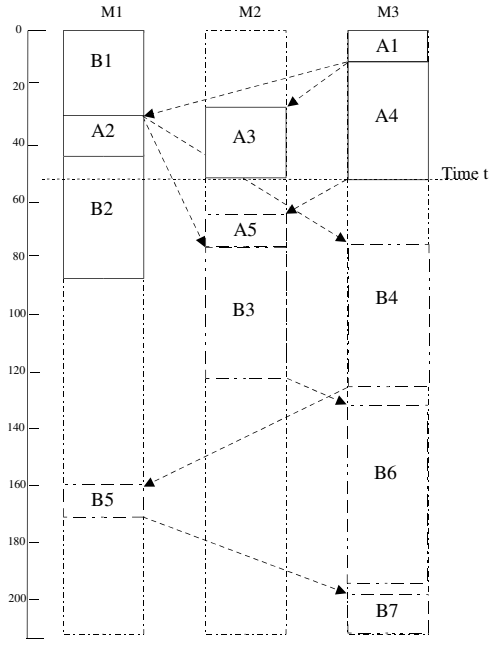


Figure 7. The schedule at time t for allocating the two given DAGs.

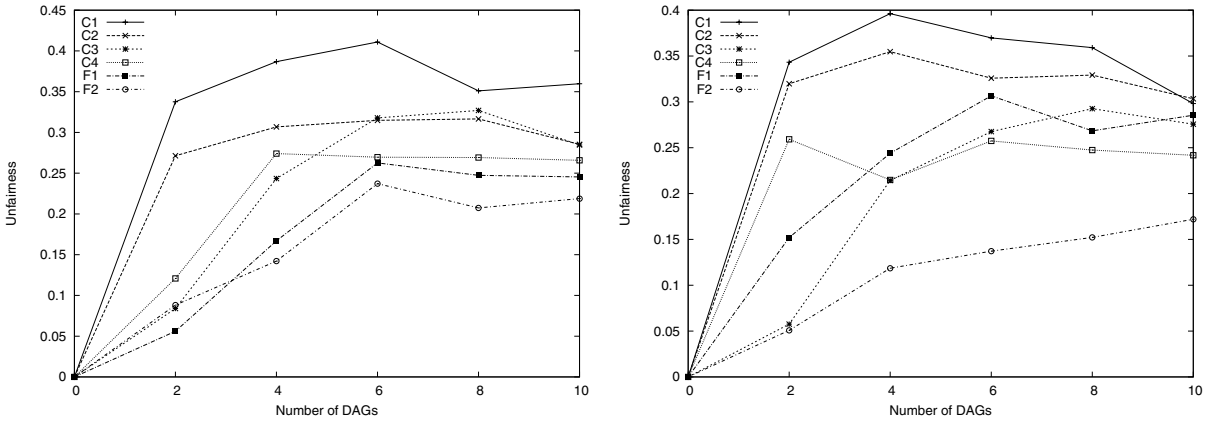
4 Experimental Results

4.1 The Setting

In this section, we evaluate the performance of the two scheduling policies aiming for fairness (denoted by F1 and F2) along with the four policies to create a composite DAG described in Section 2 (denoted by C1, C2, C3, C4). For the single DAG scheduling phase of all these policies we use Hybrid.BMCT [15] and HEFT [18]. Performance is evaluated on the basis of four metrics: unfairness value, schedule length, machine utilization and running time. We use four different types of DAGs, random (generated as explained in [21]), Laplace, Fork-Join, and FFT commonly used in other similar studies (see [16, 15]). In each case, we randomly generate 2-10 DAGs, each DAG consisting of approximately 10 to 50 tasks (depending on the type of the DAG used), and there are 3 to 8 different heterogeneous machines. For each task in the DAGs, the estimated execution time on each different machine is randomly generated from a uniform distribution in the interval 50 to 100 time units, while the communication-to-computation ratio (CCR) is also randomly chosen from the interval 0.1 to 1.

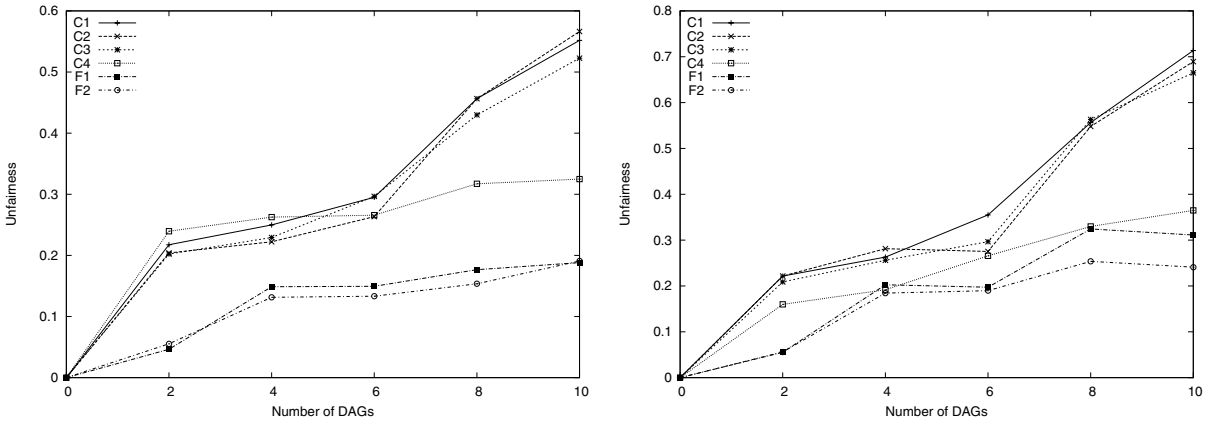
4.2 Fairness

The graphs comparing unfairness for the six scheduling approaches are shown in Figure 8, Figure 9, Figure 10 and Figure 11, for randomly generated, Laplace,



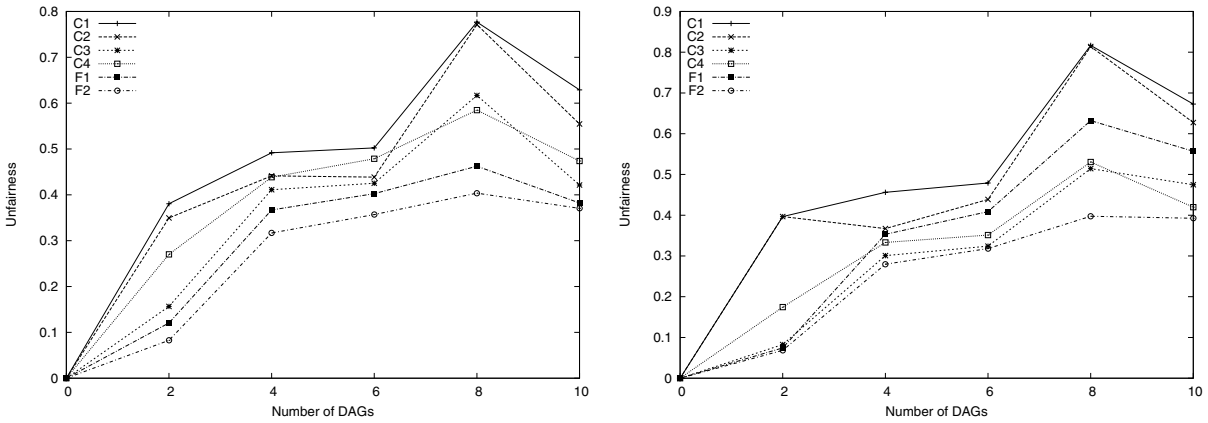
(a) Hybrid.BMCT (b) HEFT

Figure 8. Unfairness value, averaged over 100 runs, comparing 6 different scheduling techniques and two single DAG scheduling algorithms with 2-10 randomly generated DAGs, each containing 10-50 tasks, running on 3-8 machines.



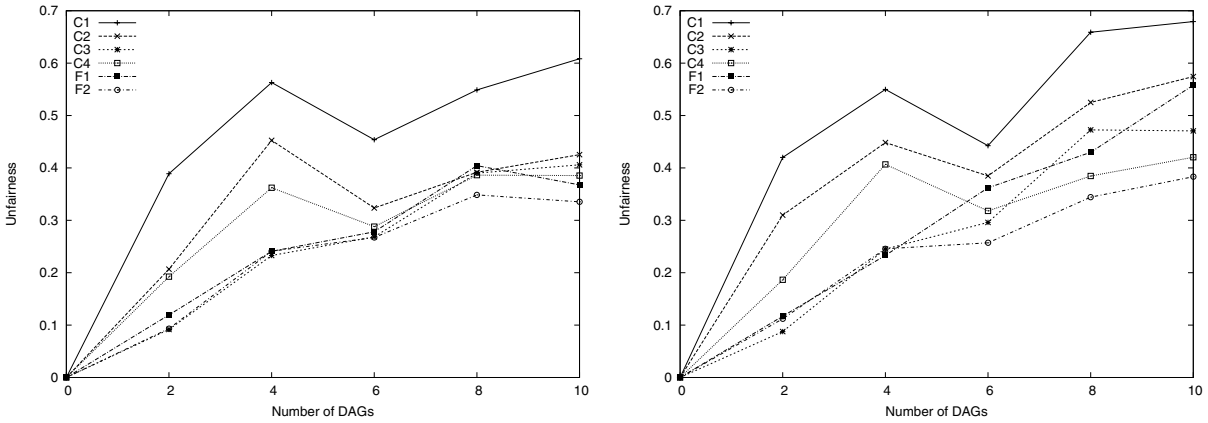
(a) Hybrid.BMCT (b) HEFT

Figure 9. Unfairness value, averaged over 100 runs, comparing 6 different scheduling techniques and two single DAG scheduling algorithms with 2-10 Laplace DAGs, each containing 9-49 tasks, running on 3-8 machines.



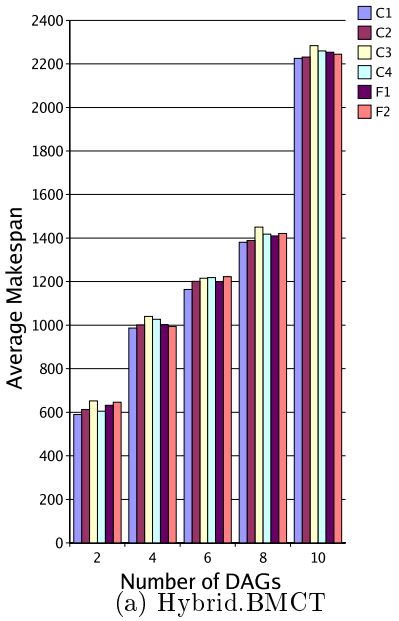
(a) Hybrid.BMCT (b) HEFT

Figure 10. Unfairness value, averaged over 100 runs, comparing 6 different scheduling techniques and two single DAG scheduling algorithms with 2-10 Fork-Join DAGs, each containing 7-44 tasks, running on 3-8 machines.

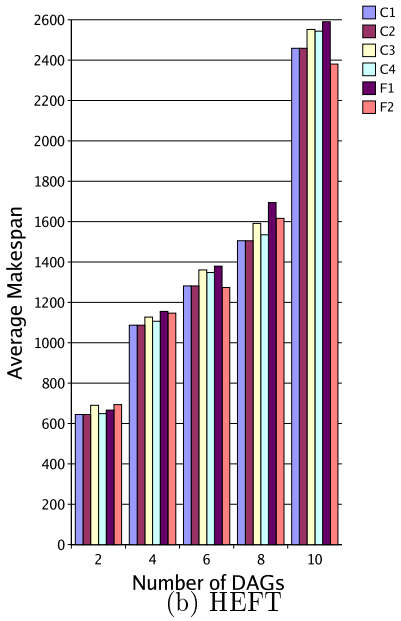


(a) Hybrid.BMCT (b) HEFT

Figure 11. Unfairness value, averaged over 100 runs, comparing 6 different scheduling techniques and two single DAG scheduling algorithms with 2-10 FFT DAGs, each containing 15-40 tasks, running on 3-8 machines.

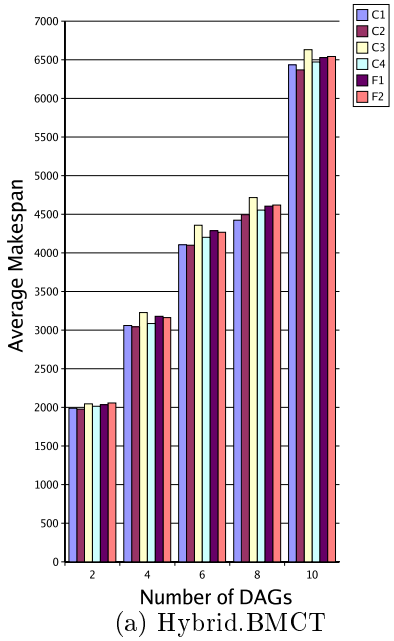


(a) Hybrid.BMCT

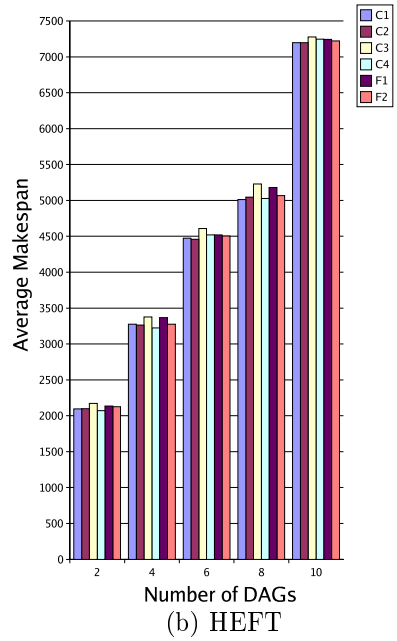


(b) HEFT

Figure 12. Average makespan (over 100 runs with 2-10 randomly generated DAGs on 3-8 machines) of six different techniques with two single DAG scheduling algorithms.

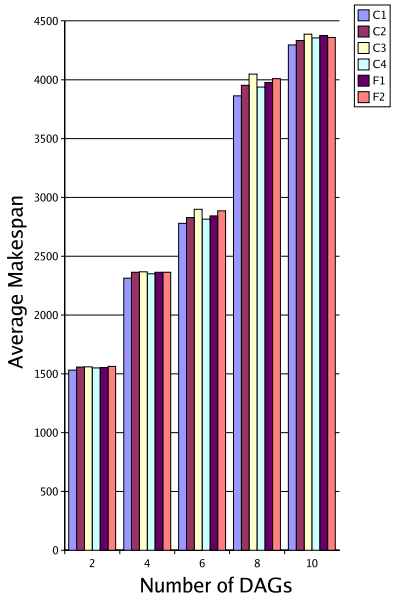


(a) Hybrid.BMCT

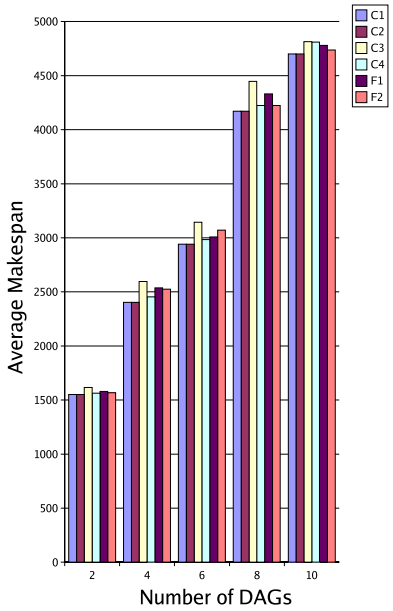


(b) HEFT

Figure 13. Average makespan (over 100 runs with 2-10 Laplace DAGs on 3-8 machines) of six different techniques with two single DAG scheduling algorithms.

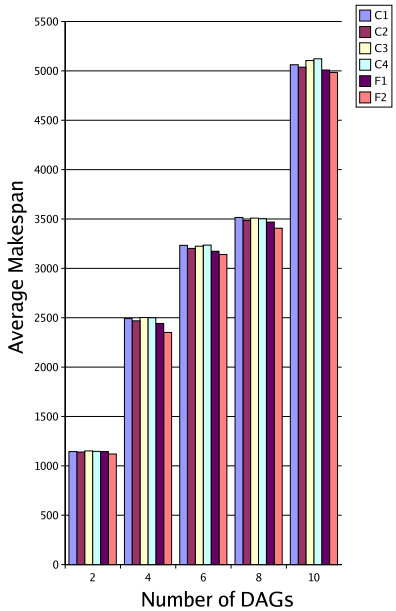


(a) Hybrid.BMCT

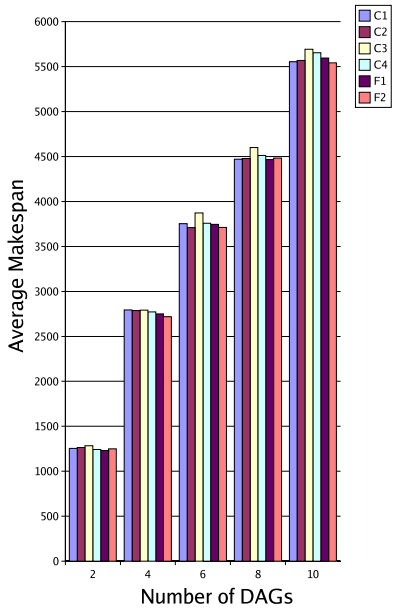


(b) HEFT

Figure 14. Average makespan (over 100 runs with 2-10 Fork-join DAGs on 3-8 machines) of six different techniques with two single DAG scheduling algorithms.

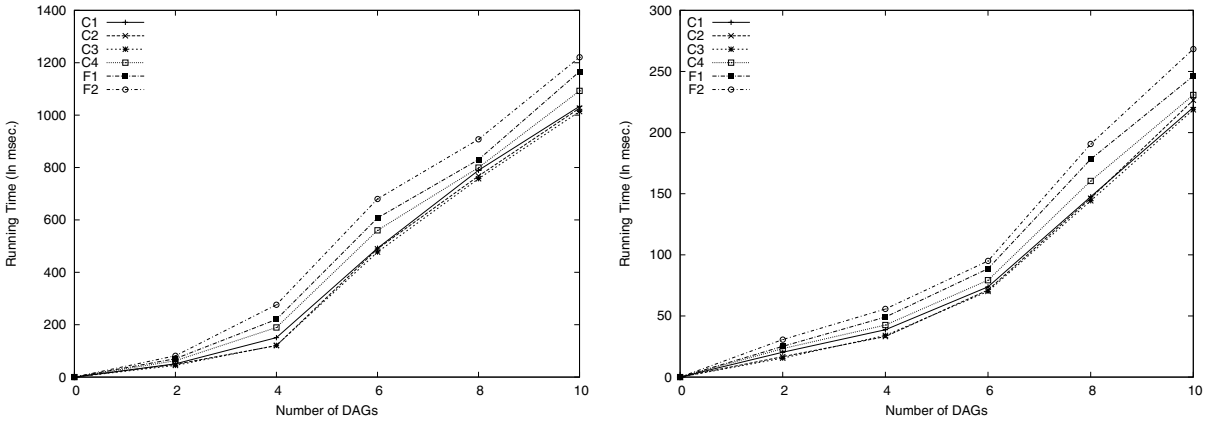


(a) Hybrid.BMCT



(b) HEFT

Figure 15. Average makespan (over 100 runs with 2-10 FFT DAGs on 3-8 machines) of six different techniques with two single DAG scheduling algorithms.



(a) Hybrid.BMCT (b) HEFT

Figure 16. Average running time of each scheduling policy over 100 runs on Randomly Generated DAGs using Hybrid.BMCT and HEFT.

	F1	F2	C1	C2	C3	C4
Hyb.BMCT	95.3	96.5	92.2	89.6	93.7	88.2
HEFT	94.2	95.7	92.1	90.1	92.0	88.6

Table 3. Average percentage (over 100 runs) of resource utilization with 4 Random DAGs, each containing 10 to 50 tasks, on 6 machines.

Fork-join and FFT graphs, respectively. From the graphs it can be seen that the schedules generated by F2 generally outperform other policies with both Hybrid.BMCT and HEFT algorithms. F1's performance is similar to F2 when the Hybrid.BMCT algorithm is used; however, it does not perform well with HEFT in most cases. Instead, C4 perform well with HEFT algorithm in Laplace and Fork-join graphs. An interesting remark from the figures is that C1, which is shown in Section 2 with the shortest makespan, is the worst method in terms of fairness (that is, the most unfair).

4.3 Makespan

The schedule length for the corresponding examples presented above is shown in Figure 12, Figure 13, Figure 14 and Figure 15, respectively. It can be seen that the makespan of the two variants for fairness is competitive to the makespan of the four composition techniques. In particular, F2 outperforms all other methods when FFT graphs are used. In addition, C3 and C4, which have reasonably good performance in terms of fairness, have generally longer makespan than F1 and F2.

4.4 Resource Utilization

In this case, we examine resource utilization, defined as a percentage of the machine time used for a task during the overall execution; results for random DAGs only are shown in Table 3. F1 and F2 seem to outperform other techniques.

4.5 Running Time

The average running time to execute our scheduling policies, averaged over 100 runs using randomly generated DAGs with Hybrid.BMCT and HEFT, is shown in Figure 16. The two policies based on fairness are slightly slower, whereas the running time appears to grow linearly with the number of DAGs being considered at the same time.

5 Conclusion

This paper considered the problem of scheduling multiple DAGs onto heterogeneous machines at the same time. A number of scheduling methods were presented; their focus was not only to minimize the overall makespan, but to achieve fairness. It was experimentally demonstrated that it is possible to achieve fairness based on an equal distribution of the slowdown amongst the DAGs, without this happening at the expense of the overall makespan. We think that this paper provides an initial only study of heuristics for multiple DAG scheduling. Further research is needed

to consider different definitions of fairness as well as other notions of Quality of Service.

References

[1] O. Beaumont, V. Boudet, and Y. Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *11th Heterogeneous Computing Workshop*, 2002.

[2] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Resource Allocation Strategies for Workflows in Grids In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*.

[3] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.

[4] F. J. Cazorla, A. Ramirez, M. Valero, P.M.W. Knijnenburg, R. Sakellariou, E. Fernandez. QoS for High Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4), July-August 2004, pp. 24-31.

[5] H. Chen and M. Maheswaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems. *International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 15-19 April 2002, Fort Lauderdale, USA.

[6] B. Cirou and E. Jeannot. Triplet: A clustering scheduling algorithm for heterogeneous systems. In *International Conference on Parallel Processing, Workshop on Metacomputing Systems and Applications*, pp. 231-236, 2001.

[7] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *Proceedings of 7th Heterogeneous Computing Workshop (HCW 98)*, March 1998, Orlando, USA.

[8] D. Jackson, Q, Snell, and M. Clement. Core Algorithms of the Maui Scheduler. *7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2001)*, pages 87–102, Cambridge, USA, June 2001.

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

[10] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu and L. Johnsso n. Scheduling Strategies for Mapping Application Workflows onto the Grid. In *IEEE International Symposium on High Performance Distributed Computing (HPDC 2005)*, 2005.

[11] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. A Realistic Network/Application Model for Scheduling Divisible Loads on Large-Scale Platforms. *International Parallel and Distributed Processing Symposium (IPDPS05)*, CD-ROM/Abstracts Proceedings, 4-8 April 2005, Denver, USA.

[12] A. Radulescu and A.J.C. van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6), pp. 648-658, June 2002.

[13] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, May 2000.

[14] G. Sabin, G. Kochhar, and P. Sadayappan. Job Fairness in Non-Preemptive Job Scheduling. In *International Conference on Parallel Processing (ICPP 2004)*, 15-18 August 2004, Montreal, Canada.

[15] R. Sakellariou and H. Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proceedings of 13th Heterogeneous Computing Workshop (HCW 2004)*, 26-30 April 2004, Santa Fe, New Mexico, USA.

[16] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4), December 2004, pp. 253-262.

[17] U. Schwiegelshohn and R. Yahyapour. Fairness in parallel job scheduling. In *Journal of Scheduling*. 3(5):297–320, 2000.

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

[19] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47:8–22, 1997.

[20] M. Wieczorek, R. Prodan and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. In *SIGMOD Record*, volume 34(3), September 2005.

[21] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par 2003*. Springer-Verlag, LNCS 2790, 2003.

Biographies

Henan Zhao received her BSc degree in Software Engineering from the University of Manchester, UK, in 2001. She is currently a PhD student in the School of Computer Science, the University of Manchester. Her research interests include task scheduling for heterogeneous and Grid environments, workflow scheduling, reservation scheduling, and algorithm design.

Rizos Sakellariou received his PhD degree in Computer Science from the University of Manchester

in 1997. Since January 2000, he has been a Lecturer in Computer Science at the University of Manchester. Prior to his current appointment, he was a visiting Assistant Professor at the University of Cyprus (Fall 1999), and a postdoctoral research associate at Rice University (1998-1999) and the University of Manchester (1996-1998). His primary area of research is in parallel and distributed systems, but his research interests also include compilers, computer architecture, performance modelling and evaluation, scheduling, and the interactions between them.