18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

Performance evaluation and optimization of an adaptive scheduling approach for dependent grid jobs with unknown execution time

Chtepen, M. [1], **F.H.A. Claeys** [2], **B. Dhoedt** [1], **F. De Turck** [1], **P.A.Vanrolleghem** [3], **and P. Demeester** [1]

[1] *Department of Information Technology, Ghent University, Sint-Pietersnieuwstraat 41, Ghent, Belgium*
Email: {maria.chtepen, bart.dhoedt, filip.deturck}@intec.ugent.be
[2] *MOSTforWATER N.V. Koning Leopold III-laan 2, 8500 Kortrijk, BELGIUM*
Email: fc@mostforwater.com
[3] *modelEAU, Département Génie Civil Pavillon Pouliot, Université Laval Québec G1K 7P4, QC, Canada*
Email : Peter.Vanrolleghem@gci.ulaval.ca

Abstract: Most nowadays scheduling algorithms for grids are based on the assumption that the application (job) execution time is known before job run-time. This assumption significantly simplifies job-resource matchmaking, although it has proven to be inapplicable for most real-world applications. On the other hand, there exists a group of applications for which execution progress can be monitored at run-time. Often, when a correlation between job execution progress and total job execution time exists, progress information can serve as a good basis for the prediction of the remaining execution time.

Another important issue in the domain of distributed computing is scheduling of jobs composed of tasks with input dependencies, whereby some tasks require inputs generated by other tasks. Since the overhead due to input dependencies is limited, this type of dependencies forms a potential for execution optimization by means of intelligent scheduling of dependent tasks on distributed resources.

In this article a detailed performance evaluation and optimization is provided for an adaptive scheduling algorithm for grids that was proposed earlier. The algorithm operates on jobs with input inter-dependencies, whose sub-tasks are organized into a DAG (Directed Acyclic Graph) and for which no information of total execution time is available. The idea behind the approach is that parallel tasks (*parent* tasks), generating input for the same underlying set of tasks (*dependent* tasks), should finish more or less simultaneously. Since the *dependent* tasks can only be executed after all the required inputs are available, the longest *parent* task is assigned to the fastest available resource, while shorter tasks can be assigned to slower resources, as long as it does not prolong the execution time of the *parent set* as a whole. The latter creates a possibility for other tasks requiring fast processing to be executed on faster machines. At first, tasks are assigned randomly. Later, the algorithm reacts on dynamic changes in resource status and variations in task execution time predictions by possibly rescheduling parallel tasks.

The algorithm's performance was evaluated using workload originating from an existing modeling and virtual experimentation tool for environmental systems (Tornado). Results have shown that significant system overhead is introduced, in terms of additional computational and network load due to the extended checkpointing and migration mechanisms. However, this overhead is compensated by more effective processing of parallel sub-tasks, which are now occupying only resources they strictly need in order not to delay the execution of the job as a whole. In this paper we measure the overhead introduced by the algorithm on network and computational resources and compare it to the overhead of a traditional static approach. It is clear that the effectiveness of the adaptive approach strongly depends on the degree of parallelism of sub-tasks and on their overall execution time heterogeneity. The boundaries for both parameters are studied. Furthermore, the performance of the algorithm can be improved by postponing migration in cases where the benefit of rescheduling is expected to be sufficiently low. Definition of the boundary for the migration postponement is also addressed.

Keywords: *Grid computing, job dependencies, performance optimization, adaptive scheduling, execution time estimation*

Chtepen *et al.*, Performance evaluation and optimization of an adaptive scheduling approach for dependent grid jobs with unknown execution time

1. INTRODUCTION

Grids are highly distributed computational environments composed of heterogeneous and de-centrally managed resources that are used for parallel execution of application (job) tasks. To make efficient use of these complex systems, a well thought through scheduling approach is essential. There are, however, several issues that make the quest for an appropriate scheduling solution far from trivial: grids are dynamic environments, with constantly changing resource load and availability; exact execution times of jobs running on a grid are often unknown in advance; possible dependencies between tasks composing a job require a certain execution sequence, *etc*. Most of the currently existing scheduling algorithms for grids (Chang, 2007; Rahman, 2007) make a simplifying assumption that grid and/or workload parameters are known *a priori*. The latter, however, significantly reduces the practical applicability of the proposed methods.

Fortunately, in many applications there exists a correlation between job progress and total job execution time. This means that the remaining task execution time can be estimated when the task progress on a particular resource and the current execution time can be monitored. Therefore, in our previous work we proposed an adaptive scheduling algorithm that reassigns (reschedules) tasks at run-time, based on dynamically collected information on task execution progress and on the status of grid resources. The algorithm schedules relatively efficiently tasks with input dependencies, which is a type of dependencies, whereby a task can require inputs generated by other tasks in order to proceed with its execution. Particular to input dependencies is that they do not require extensive message passing between communicating tasks (a single data transfer normally suffices), which provides possibilities for execution optimization by means of intelligent assignment of dependent tasks to widely distributed computational resources within grids.

The main disadvantage of the algorithm proposed earlier is that it introduces considerable overhead in terms of computational delay and network load due to frequent rescheduling. However, this overhead is compensated by faster processing of dependent tasks. In this paper we closely observe the algorithm's performance and compare it to the performance of a static approach. Afterwards, we observe how the overhead can be reduced by avoiding redundant rescheduling.

The remainder of this paper is organized as follows: Section 2 summarizes related work; in Section 3 the adaptive approach for scheduling of tasks with interdependencies is discussed; in Section 4 we take a close look at the performance of the adaptive approach; and finally, Section 5 concludes the paper.

2. RELATED WORK

Since knowledge of the exact task execution time offers many possibilities for efficient task resource matchmaking, various research efforts were dedicated to the design of accurate task run-time prediction mechanisms. For instance, in (Nassif, 2005) the use of historical information is proposed for this purpose. Each task is supposed to be provided with a set of descriptive attributes (e.g. application name, requirements) that identify similarities between different tasks. The execution time of a new task is determined from run-times of similar tasks executed on the same host. (Caniou, 2004) introduces another history-based approach, called Historical Trace Manager (HTM), which is designed particularly for GridRPC-based middleware. The HTM takes into account the time-shared server model and application properties (e.g. size of input and output data, the number of operations in each task) to predict the duration of a task on a particular resource and its impact on the execution of other tasks. The disadvantage of both above-mentioned approaches is that they require a significant amount of information on system performance and the executed applications. However, in dynamically changing grid environments often running complex applications with sophisticated control flows, providing such information is a challenging task. On the other hand, our approach does not require any *a priori* information, since it can dynamically modify its decisions based on application and system performance updates collected at run time.

DAGs are a commonly utilized structure for the representation of tasks with interdependencies. In (Malewicz, 2006) a DAG-based scheduling tool was developed that assigns task priorities to maximize the number of *eligible* tasks in each step of the computation. An *eligible* task is a task whose parents have already been processed and which is ready to execute in the next scheduling round. Opposite to this approach, our adaptive algorithm tries to reduce the execution time of a job as a whole, which can eventually be achieved by slowing down the execution of individual tasks. Yet another DAG-based approach is introduced in (Aggarwal, 2005): tasks are organized into a DAG-structure according to the constraints imposed by the end-user. A Generic Algorithm based scheduler optimizes resource utilization while satisfying conflicting user goals.

3. ADAPTIVE SCHEDULING ALGORITHM

In our previous work (Chtepen, 2008) an adaptive scheduling approach was introduced, whose functionality is shortly summarized in this section. The proposed approach operates on tasks with input interdependencies, for which the total execution time is unknown in advance but gradually improving execution time estimates can be computed at run-time. The objective of the algorithm is to speed up the execution of a job as a whole, at the cost of partially slower execution of individual tasks.

The algorithm operates on tasks organized into a DAG structure, as shown in Figure 1, where circles indicate tasks belonging to the same job and edges represent input dependencies. The considered DAG graph represents a job dependency structure often occurring in real-world application. As shown on the figure, we assume a single *initial* and a single *final* task. The *initial* (*parent*) task can have an arbitrary number of *dependents* (e.g. simulations with varying parameter values), outputs of which can be combined or split to serve as inputs for *dependents* on the next level of the graph. The considered DAG can have an arbitrary height, *i.e.* an arbitrary number of levels with *dependent* tasks. At the final level, all intermediate results are combined for input into the *final* task, where the final job output is calculated.

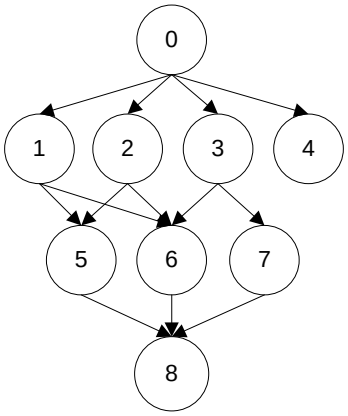

Figure 1. Example of a job composed of tasks with input dependencies, organized into a DAG structure.

The idea behind the algorithm is to schedule parallel tasks having the same *dependents* (the so-called *parent set*), in such a way that they finish more or less simultaneously. In Figure 1, the set {1,2} is an example of a *parent set* generating inputs for *dependents* (5 and 6). Since tasks within a *parent set* often have varying length and *dependents* require inputs from the whole *parent* set to proceed with their execution, it is beneficial to assign short tasks within the *parent set* to slower resources. Faster resources that become available can in turn be occupied by longer tasks requiring faster execution. It is important to notice that a *parent set* can contain multiple *parent subsets* ({1,2} is a subset of {1,2,3}), which execution is optimized *prior* to the execution optimization of the incorporating *parent set*. The operation of the adaptive algorithm can be subdivided into the following three steps (see Figure 2):

Step 1 (dynamic collection of information): run-time information on resources status, on the progress of running jobs and on the location of all *initial* tasks is collected. The latter is particularly important in heavily utilized grids with a limited number of computational resources. Since *initial* tasks do not have input dependencies, they can be started immediately, occupying to a large extent the available resources and delaying the execution of *dependent* tasks. Therefore, it should be possible to rapidly locate and interrupt / reschedule *initial* tasks when a *dependent* task is ready to run.

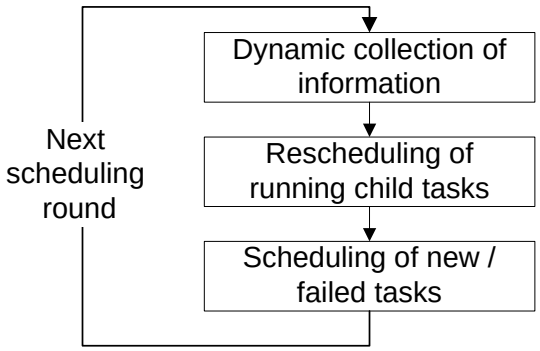

Figure 2. Flow of the operation phases of the proposed adaptive scheduling algorithm.

Step 2 (rescheduling of parent sets): this step is executed if some fully scheduled *parent sets* (with no idle tasks) are running on the grid. Tasks within such *parent sets* are reassigned as a reaction on newly available information on task progress and possibly a changed grid status (e.g. availability of new computational resources, resource failure, variations in resource load). Tasks are rescheduled in the scope of their *parent sets*. The algorithm processes *parent sets* in the order of increasing number of still running tasks in a set. This scheduling order ensures that almost completed *parent sets* are scheduled to the fastest available resources.

For each task within the next *parent set* (*PS*) to reschedule, the algorithm calculates a new estimation of the task execution time: $E = 100\% / P \times T$, where P is the percentage of the workload already completed and T is the elapsed execution time. Afterwards, for the task $j \in PS$ with the longest estimated execution time (E_{max}) the algorithm defines a set R of available resources that are faster than the current resource where j is running. The speed of a resource is determined as the ratio between its $MIPS_r$ (Million Instructions Per

Second) and the number of tasks it is currently executing (*initial* tasks not counted). The algorithm tries to reschedule *j* to the fastest resource $r \in R$ and here the following scenarios can occur:

- *R* is empty because *j* is already running on the fastest available resource: no rescheduling within *PS* can take place and the algorithm proceeds with the second shortest *parent set*.
- *R* is not empty but *j* can not be reassigned to *r* without delaying the execution of other *parent sets*: the algorithm deletes *r* from *R* and tries to reschedule *j* to the second, third, *etc*. fastest resources.
- *j* is reassigned to *r*, and the new maximum estimated execution time (E_{max}^{new}) over all task in *PS* now belongs to another task *i*: the reassignment procedure is repeated again for the task *i*.
- *j* is assigned to *r*, but it still has the longest estimated execution time (E_{max}^{new}) over all tasks in *PS*: the current limit on the *PS* execution time reduction is reached. The algorithm proceeds by linking other tasks $i \in PS$, $i \neq j$ to the available resources in such a way that the difference ($E_{max}^{new} - E_i$) is minimized and $E_i < E_{max}^{new}$. This implies that the remaining tasks from *PS* are reassigned to the slowest possible resources that keep the maximum estimated execution time within *PS* below E_{max}^{new}. This procedure, however, is not applied to tasks $i \in PS$ that are part of another *parent set* $PS_s \in PS$, because it would delay the execution of the shorter PS_s set.

Each time a *dependent* task is rescheduled to a particular resource, the *initial* tasks running on that resource can be interrupted and reassigned to the slowest available free resources.

Step 3 (scheduling of new/failed tasks): in this part of the algorithm, ready-to-execute tasks (*i.e*. tasks with either no *parents* or all *parents* already executed) are assigned to the resources that are still available after the rescheduling phase. If resource availability allows, the algorithm attempts to process tasks belonging to the same *parent set* in a single iteration. Tasks are processed in the order of their arrival into the system (longest waiting task first), however partially processed *parent sets* always get the highest priority. Task-resource matchmaking proceeds to a large extent similar to Step 2, except that idle tasks within *parent sets* are assigned to the resources taking into account their rough initial length estimation. For failed tasks, the predicted length is eventually reduced with the already performed and checkpointed computations.

4. PERFORMANCE EVALUATION

The performance of the proposed adaptive scheduling algorithm is evaluated using the workload parameters collected from the existing modeling and virtual experimentation tool for environmental systems, named Tornado (Claeys, 2006). In Tornado, two types of input dependencies typically occur:

- **Parameter sweep**: sub-experiments (Tornado equivalent for a task) are run on the same model but with varying parameters, which results in similar execution times.
- **Model sweep**: sub-experiments are run on different models with strongly varying execution times.

In this section we evaluate the performance of the adaptive approach for both above-mentioned types of task interdependencies. We observe to what extent computational overhead and network load are influenced by execution time heterogeneity and by the degree of parallelism between different sub-experiments.

Furthermore, characteristic to Tornado jobs is that their execution time is extremely difficult to predict in advance, because of the dynamics of the enviromental processes that are being modeled. However, for experiments with a predefined simulation horizon, the execution progress can easily be calculated at run-time, using the equation $E = 100\% / P \times T$, where $P = T_s^c / T_s$. T_s^c stands for the current simulated time and T_s

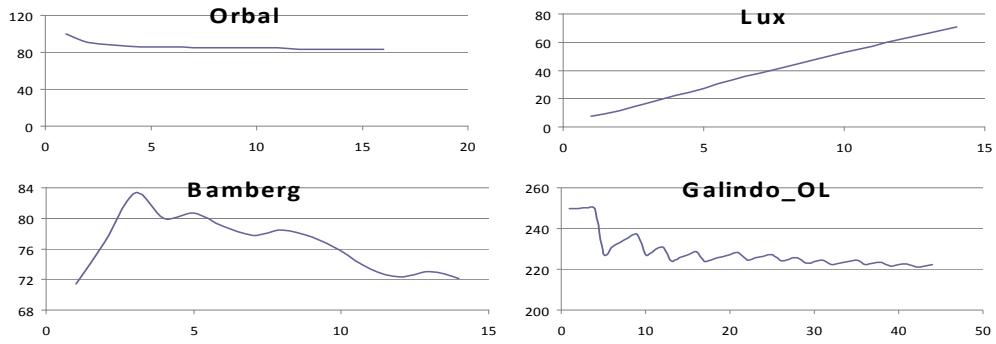

Figure 3. Examples of Tornado simulation experiments following *decreasing* (Orbal), *increasing* (Lux) and *oscillating* (Bamberg and Galindo_OL) execution time prediction models.

is the total simulated time. Based on the monitored progress we try to predict at each measurement point the total execution time of a Tornado sub-experiment. This prediction can strongly vary over time, which leads to the notion of an execution time prediction model. This model shows the evolution of the predicted job length over time. In our previous work (Chtepen, 2008), all prediction models were subdivided into 3 categories: *decreasing*, with execution time estimations gradually decreasing over time; *increasing*, with execution time estimations gradually increasing over time; and *oscillating*, with execution time predictions oscillating over time (see Figure 3). When a preliminary comparison between the adaptive scheduling algorithm and a simple static approach that schedules arriving jobs randomly was performed, the results suggested that the advantage of the dynamic method, compared to the static one, is most significant for execution time estimates following the *increasing* model. This can be explained by the fact that in the latter case we have to deal with relatively long tasks, on which overhead due to occasional checkpointing and migration has a rather small impact. Therefore, in what follows we consider only the *increasing* model, which will provide the upper boundary on the adaptive algorithm performance.

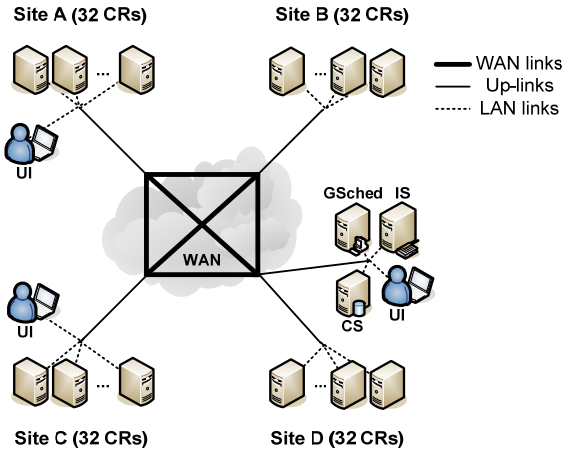

Figure 4. Example grid architecture: *UI* (User Interface), *GSched* (Grid Scheduler), *IS* (Information Service), *CS* (Checkpoint Server), *WAN* (Wide Area Network), *LAN* (Local Area Network).

For our validation experiments we have modeled the grid environment shown in Figure 4. It consists of 4 widely distributed computational sites, aggregating all together 128 computational resources (*CR*'s), connected within the sites by Local Area Networks (LANs) with start topology. Furthermore, the grid contains a number of general services: a user interface (*UI*), through which tasks are submitted into the system; a scheduler (*GSched*) responsible for task-resource matchmaking; an information service (*IS*), which collects task and resource status information required by the GSched; and a checkpoint server (*CS*) where checkpointing data is made persistent. It is assumed that all the CR's as well as the general services are fully stable. That means that no failures can occur and that load on each resource remains constant over time. In compliance with real-world grid deployments, CR's in our model possess different computational power that varies from 100 to 4000 MIPS. Finally, the Wide Area Network (WAN), connecting the sites, has a bandwidth of 1 GBit/s and a latency varying from 3 to 10 ms, while the intersite LAN-links have a bandwidth of 1 Gbit/s and a latency of 1ms.

The assumed job submission model simulates the behavior of typical Tornado users. Jobs are arriving into the system in batches of varying length with a frequency that follows a daily cycle. The latter implies that

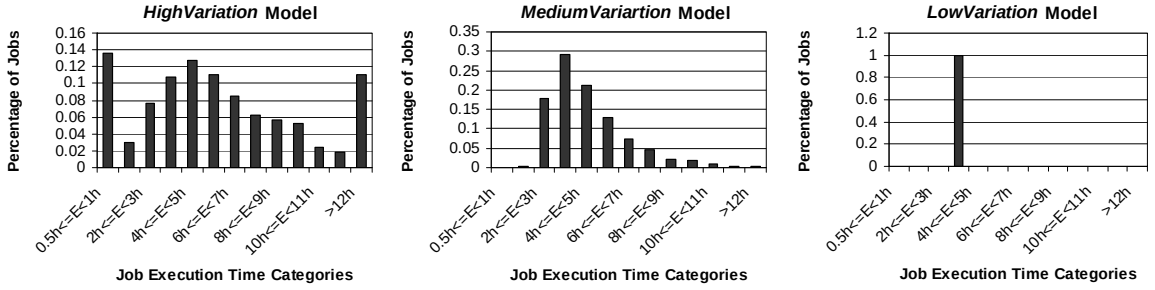

Figure 5. Job execution time distributions for *HighVariation*, *MediumVariation* and *LowVariation* models.

most of the jobs (about 80%) arrive during day-time. Three different models were considered to represent variations in task executions times: *HighVariation*, *MediumVariation* and *LowVariation*. Task lengths derived according to these models are shown in Figure 5. To simplify the comparison between different models it is assumed that the size of inputs, outputs and checkpoints amounts 1 MB for all tasks; and a task checkpointing delay varies from 100 ms to 1 s, depending on the actual execution time. The limit for the increase in job execution time is initialized to 200 % of the initially estimated job length. The algorithm performance is compared for varying degree of parallelism of sub-tasks, whereby we consider *parent sets*

with 2, 5, 10 and 20 parallel tasks within DAG graphs consisting of 3 levels. The grid described above was observed during 24 hours of simulated time.

Figure 6 compares the efficiency of the proposed adaptive approach to a random static approach, in terms of the amount of useful workload processed. As "useful workload" we consider tasks belonging to successfully

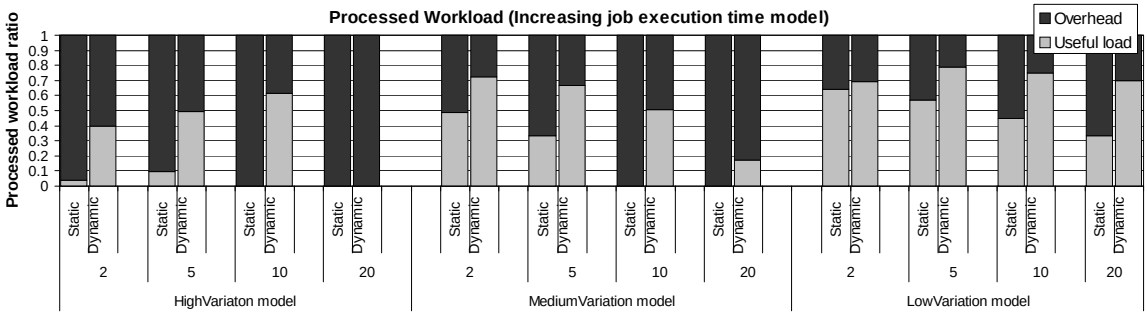

Figure 6. Proportion of useful workload processed by the dynamic and the static approaches for increasing job execution time model.

executed final jobs. The results show hat the lower the parallelism and the length variation between executing tasks, the smaller the difference between both approaches. The use of the dynamic algorithm is the most advantageous when scheduling jobs with a high task length variation and a somewhat intermediate degree of parallelism. This can be explained by the fact that when only a few tasks with similar lengths are executed in parallel, the gain due to intelligent scheduling is rather limited. On the other hand, when execution times of large number of parallel tasks have to be balanced, the balancing procedure becomes lengthy and the difficulty of finding a sufficient amount of suitable resources arises. Worth noticing is that in optimal circumstances the dynamic algorithm can outperform the static one by as much as 60%.

Figure 7 shows the network load introduced by the dynamic and the static algorithms. The dynamic approach

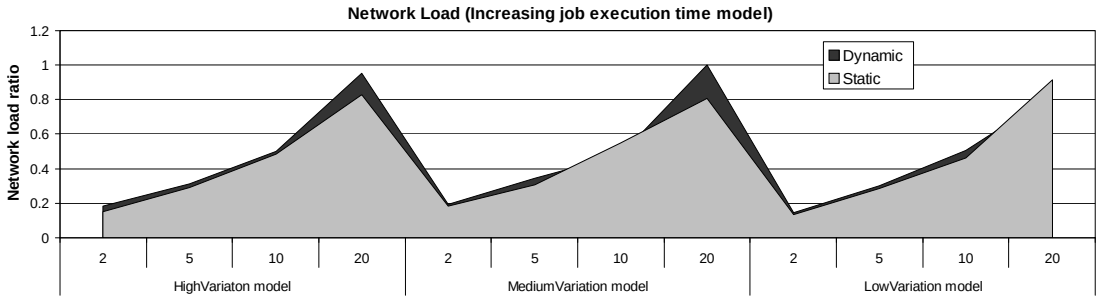

Figure 7. Network load of the dynamic and the static approaches for the increasing job execution time model.

introduces the most network overhead due to task reassignments (up to 20% more than the static algorithm) when trying to calibrate the execution times of a large number of parallel tasks.

Obviously, the exact network overhead, as well as the amount of useful workload processed, strongly depends on the cost of a migration, *i.e.* complexity and size of checkpoints, size of input and output data. Sometimes the performance of the adaptive approach can be improved by reducing the number of migrations performed. Therefore, we modified the dynamic algorithm to perform migration of *dependents* only when rescheduling reduces the predicted execution time with a certain minimum percentage (*MP*). Figure 8 gives an overview of the achieved results when comparing the original solution with the solution where *MP* is initialized with 20, 50 and 70 %. We observe only the case with parent sets containing 5 parallel tasks, since in other cases the algorithm performs more or less similar. From the results can be concluded that for the given scenario the *optimal* performance (about 10% improvement) of the adaptive algorithm is achieved when migrations resulting in relatively small execution time improvements (less than 20%) are skipped. However, omissions of migrations leading to higher execution time improvements are not longer compensated by reduction in migration overhead. Therefore, the system performance starts to degrade. Since

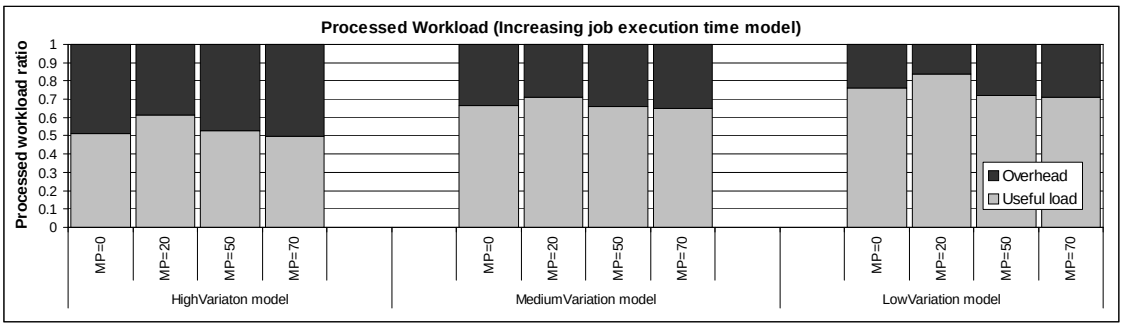

Figure 8. Fraction of useful workload processed by the dynamic approach with varying MP-values.

the optimal frequency of migration is strongly system and application dependent, the latter can best be determined and calibrated based on historical information on previous application runs.

5. CONCLUSIONS

Scheduling of dependent tasks in distributed and dynamic grids is an important issue, especially because the exact job execution times can hardly be determined in advance. Therefore, in our previous work an adaptive scheduling approach was introduced that reschedules tasks at run-time based on monitoring of grid status and task execution progress. In this paper, a detailed performance evaluation of the algorithm was performed. Where in our previous work we compare the adaptive and static approaches in terms of their average job turn-around times and the number of jobs processed, this paper mainly focuses on computational and network overhead of the adaptive approach under different circumstances. Furthermore, we propose a remedy for overzealous migration, which is the main cause of the overhead of the adaptive approach. An improved version of the algorithm skips task rescheduling when it leads to an insufficient performance improvement. Simulation results have shown that this method can lead to significant performance gain, however depending on the migration costs of each particular application.

REFERENCES

Aggarwal, M., R. Kent, and A. Ngom, (2005), Genetic Algorithm Based Scheduler for Computational Grids. In Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications, Guelph, Ontario, Canada, 15 – 18 May 2005.

Caniou, Y., and E. Jeannot, (2004), Experimental Study of Multi-Criteria Scheduling Heuristics for GridRPC Systems. In Proceedings of ACM/IFIP/IEEE Euro-Par-2004: International Conference on Parallel Processing, Pisa, Italy, 31 August – 3 September 2004.

Chang, R.S., J. Chang, and P.S. Lin, (2007), Balanced Job Assignment Based on Ant Algorithm for Computing Grids. In Proceedings of the 2nd IEEE Asia-Pacific Services Computing Conference, Tsukuba, Japan, 11 – 14 December 2007.

Chtepen, M., F.H.A. Claeys, B. Dhoedt, F. De Turck, P.A. Vanrolleghem, and P. Demeester, (2008), Scheduling of Dependent Grid Jobs in Absence of Exact Job Length Information. In Proceedings of the 4th IEEE/IFIP International Workshop on End-to-end Virtualization and Grid Management, Samos Island, Greece, 22 – 26 September 2008.

Claeys F., D. De Pauw, L. Benedetti, I. Nopens, and P. Vanrolleghem, (2006), Tornado: a Versatile and Efficient Modelling & Virtual Experimentation Kernel for Water Quality Systems. In Proceedings of the International Environmental Modelling and Software Conference, Burlington, VR, July 9–13 2006.

Malewicz, G., I. Foster, A. Rosenberg, and M. Wilde, (2006), A Tool for Prioritizing DAGMan Jobs and Its Evaluation. In Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing, Paris, France, 19 – 23 June 2006.

Nassif, L., J. Nogueira, M. Ahmed, A. Karmouch, R. Impey, and F. Vinicius de Andrade, (2005), Job Completion Prediction in Grid Using Distributed Case-based Reasoning. In Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise, Linköping, Sweden, 13 – 15 June 2005.

Rahman, M., S. Venugopal, R. Buyya, (2007), A Dynamic Critical Path Algorithm for Scheduling Scientific Workflow Applications on Global Grids. In Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing, Bangalore, India, 10 – 13 December 2007.