# Dependency Patterns and Timing for Grid Workloads

Andrew Burkimsher

Department of Computer Science, University of York

## Abstract

This paper presents a set of patterns of dependencies for grid computing workloads abstracted from an industrial case study. In addition, algorithms are presented that generate task execution times and arrival times to match desired statistical properties. This is as a part of the research performed by the author on the creation of a simulation environment with which to compare and evaluate the performance of different schedulers on a grid system. The ultimate aim of this research is to improve throughput and reduce response times of work submitted to a grid using improved scheduling algorithms.

## 1 Introduction

Computing power has become ever cheaper over the past half-century following an observed pattern known as Moore's Law [14]. However, many kinds of academic and industrial endeavour benefit from as much computing power as possible. In recent years, increases in computing power have been gained through increasing parallelism [5].

Where there is sufficient demand for compute power, datacentres have been created to house these parallel computing nodes [13]. Unfortunately, datacenters are limited in size by the availability of electrical power and cooling in a single location [15]. Therefore, networks of geographically distributed datacenters have been created in order to provide the computing capacity required [11]. These networks are known as *grids*. In the model considered by this paper, many workloads may execute concurrently on a grid. In order to ensure good performance of a grid, the work done must be carefully scheduled.

It has been shown that finding an optimal schedule is NP-complete in the general case [9]. Therefore in any system of realistic scale, heuristic scheduling policies have to be used. Where dependencies are present in the workload, some 'simple' scheduling policies such as *First In First Out* (FIFO) exhibit undesirable emergent effects known as 'anomalies' [10,17]. [10] showed that reducing the number of dependencies, reducing task execution time and increasing the number of processors can all lead to an increase in total workload execution time when using a FIFO scheduler. [17]showed that network delays can mean that executing a workload on a multiprocessor would take longer than on a uniprocessor.

A wide variety of scheduling heuristics have been proposed [2]. The ultimate aim of this research is to develop a framework with which to simulate grids in order to compare the performance of schedulers. The simulation will be composed of three fundamental models: an application model, a platform model and a scheduling model. In this structure, the grid hardware is represented by the platform model. The workload executed by the grid is represented by the application model. The scheduling algorithm is captured in the scheduling model.

This paper is about the creation of workloads as part of the application model. Pertinent literature will be summarised. The DAG shape patterns used in taskset generation, along with algorithms to generate them will be presented. Algorithms to generate workloads where both jobs and task execution times follow desired statistical distributions will be proposed. Finally, algorithms will be proposed to adjust the arrival rate of the work in reference to the ability of a grid to service the work (stability ratio), while keeping the workload otherwise equal. Limitations of the work along with future research directions will be noted.

## 2 Literature Review

To utilise the computational power afforded by grids, the work must be parallelised. Some kinds of computational work scale naturally to being run in a highly parallel way. However, real-world grid workloads are not like this. They tend to have sections that can be parallelised but others that must run in sequence.

Some nomenclature will now be introduced, following the scheme of [4]. Independent packages of work are known as *jobs*. Within a job, the sections of work are known as *tasks*. Each task runs on a single processor and consumes some input and produces some output. Where one task's input includes another task's output, a *dependency* is defined.

There has been much study of the scheduling of dependent task sets in the literature; notable examples include [10,12,16]. The general notion of dependencies in the literature is to consider the dependencies to be representable by a Directed Acyclic Graph ($DAG$) structure. The acyclic nature of the dependencies means that the computation time of each workload is bounded. The nodes in a DAG represent tasks and edges represent dependencies between tasks.

Although many authors have mentioned the use of a DAG structure for workloads [12], there is scant mention in the literature of the actual structure of these DAGs and how to generate them [7]. This paper will elaborate several classes of DAG structure in Section 3. These structures are abstractions of patterns observed in an industrial case study.

The dependency structure alone, however, does not define a grid workload as the execution times of tasks must also be defined. When generating large numbers of workloads to use in the comparison of schedulers, it is important that the workloads generated have certain statistical properties, so that they form a fair comparison [7]. Algorithms to generate workloads with appropriate statistical distributions of both job and task execution times will be described in Section 4.

## 3 DAG shape classes

The exact parameters of real-world workloads are unlikely to be known in advance. Therefore, a successful grid scheduling policy should be able to perform well across a wide range of workloads. To evaluate schedulers, therefore, a wide range of workloads must be generated. However, the workloads generated should also contain a fair representation on the kinds of workload likely to be encountered by the scheduler in production use.

The workload DAG patterns presented in this section were developed from an industrial case study. The industrial case study was of a production system that takes CAD models and performs computational fluid dynamic calculations with them in order to produce predictions of aerodynamic characteristics. Workflows are specified in advance, and these workflow task graphs can be inspected. After inspection, general patterns were observed by the author and are presented below. To the best of the author's knowledge, all the patterns except the linear chains pattern are novel and have not been previously described in the literature. Pseudocode algorithms for generating these DAG shapes are also specified.

### 3.1 Linear Dependencies

The most basic DAG dependency pattern is that of linear dependencies. This is when there is a single chain of purely sequential tasks with dependencies between them, as shown in Figure 1a. However, this pattern could well be considered unrealistic for a grid workload. This is because grids tend to perform best on parallel workloads, so it is highly unlikely that a substantial part of any real grid workload would be composed of linear dependent chains of work. Nevertheless, if it were, an appropriate scheduling policy could be a pipeline arrangement. The pseudocode to set up dependencies like this is shown in Algorithm 1.

### 3.2 Probabilistic dependencies

Sometimes it is desired to have a certain proportion of the possible dependencies present in a workload. If it is desired that these dependencies are randomly sampled from the set of possible dependencies, the probabilistic dependencies method can be used. The pseudocode algorithm for this is shown in Algorithm 2. Two sample task graphs are shown in Figure 1d.

This algorithm has the advantage that the shape of the dependency graph can vary significantly, and given enough samples should provide a wide variety of shapes with which to exercise a scheduler. However, there is a strong likelihood when low probabilities are used that the dependency graph for each job can have disconnected sections. By the definition given earlier, disjoint dependency graphs should really be represented as separate jobs.

Although the job could be split into two separate jobs, or have the disjoint sections connected with additional dependencies, this may interfere with the statistical properties desired in the workload. It could be possible to simply discard jobs that where the graph has disjoint parts. However, as the probability is decreased then an increasing number of jobs may be discarded, to the point where it may become impractical to generate workloads this way because too many jobs are being discarded. As the probability is increased, this method approximates the linear dependencies model (if transitive dependencies are removed). For all these reasons, this method is only really suited to probability values in the middle of the probability range.

### 3.3 Independent Chains

Many workloads are parallelised by applying the same sequence of operations to different chunks of data. Each chain is one following the linear dependencies pattern. This is inspired by the MIMD (Multiple Instruction Multiple Data) parallelism pattern. As observed in an industrial case study, these chains need to be spawned by an initial setup task. Their results are then collected up by a final task. A diagram showing this arrangement is shown in Figure 1b. Pseudocode for generating such a configuration is shown in Algorithm 3.

### 3.4 Diamond

The diamond pattern as shown in Figure 1c is similar to the independent chains model, but where the spawn-out of independent chains does not take place all at once, but requires several stages to perform. It could also be considered like a complete binary tree branching out to the maximum width, and then condensing down again to collect up the data. Pseudocode for defining these dependencies is given in Algorithm 4.

### 3.5 Dependencies over blocks

A single generation of the independent chains or diamond pattern can be considered as a *block*. A block is a subset of the tasks in a job with a single starting and a single finishing task. Workloads can be composed of dependencies between blocks. The existing patterns shown can then be extended to also cover blocks. The first and last tasks of each block are given the incoming and outgoing dependencies of the whole block. These blocks then become building blocks for more complex DAGs. Where a compositional approach is used with blocks, it becomes possible to represent arbitrary DAGs.

A prevalent shape of workload observed in the industrial case study was that of linear chains of blocks, where the blocks followed the independent chains pattern ( Figure 1e). This is observed where each stage of the workload can be parallelised, but the data between each stage may need to be collated and transmitted before the next stage of execution can commence. These patterns can be particularly challenging to schedule efficiently because of the multiple bottlenecks between the blocks. However, they are important to study when comparing schedulers, because they represent a significant fraction of the workload observed on some industrial grids.

**Algorithm 1** Pseudocode for the Linear Dependencies pattern

```
n = number of tasks
task[1].dependencies = {}
for taskid in [2 to n]
        task[taskid].dependencies = {task[taskid-1]}
```

**Algorithm 2** Pseudocode for the Probabilistic Dependencies pattern

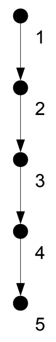
```
n = number_of_tasks
p = dependency_probability
for taskid in 1..n:
        for possible_dep_id in taskid..n:
                if p <= random():
                        tasks[taskid].dependencies.add(
                                 tasks[possible_dep_id])
```

**Algorithm 3** Pseudocode for the Independent Chains pattern

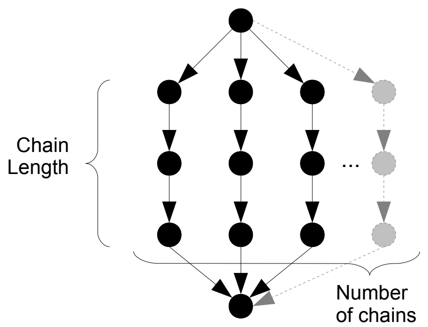
```
all_tasks = empty list of tasks
task_inner_matrix = 2-d matrix of tasks of shape
                        (num_chains by chain_length)
for x in 1..num_chains:
        for y in 2..chain_length:
                task_inner_matrix[x][y].dependencies.add(
                        task_inner_matrix[x][y-1])
                all_tasks.add(task_inner_matrix[x][y])
for x in 1..num_chains:
        task_inner_matrix[x][1].dependencies.add(initial_task)
        final_task.dependencies.add(
                task_inner_matrix[x][chain_length])
all_tasks.add(initial_task)
all_tasks.add(final_task)
return all_tasks
```

**Algorithm 4** Pseudocode for the Diamond pattern

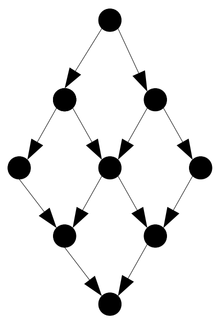
```
d = diamond_edge_length
task_matrix = 2-d matrix of tasks with dimensions d * d
for x in 1..d:
        for y in 1..d:
                if x > 1:
                        task_matrix[x][y].dependencies.add(
                                        task_matrix[x-1][y])
                if y > 1:
                        task_matrix[x][y].dependencies.add(
                                        task_matrix[x][y-1])
```



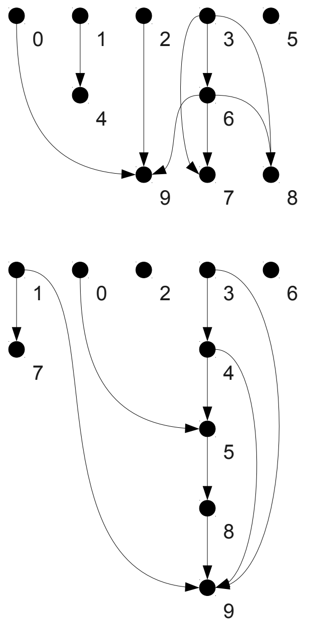
(a) Linear Dependencies



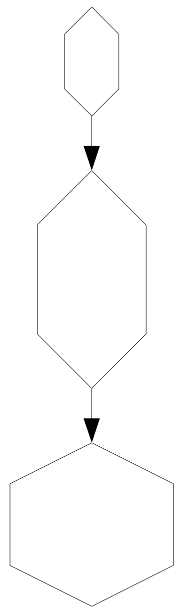
(b) Independent Chains



(c) Diamond



(d) Probabilistic Dependencies ($T = 10, P = 0.3$)



(e) Chain of Blocks

Fig. 1: Dependency DAG shapes

## 4 Execution Times and Stability

Section 3 describes the shape of the DAG of a workload's dependencies. However, the execution times of each task within the workload must also be specified, as must the arrival time of each job. The scheduler must know this information in order to be able to make appropriate scheduling decisions. This section will firstly describe ways of generating workloads with task and job execution times that conform to a desired statistical distribution. Secondly, an algorithm is described to set the arrival time for each job to ensure a given stability ratio.

### 4.1 Distribution of Execution Times

According to the model defined above, each task has a specified execution time. These task execution times need to be generated in such a way that the workload has statistically robust properties [8].

The simplest method of assigning execution times to all the tasks in the workload is simply to generate a random number in a given range for each task. However, this means that where jobs are composed of a similar number of tasks, they will also have a similar total execution time [1].

When generating many workloads that are comparable, it is highly desirable to be able to create them with the same total workload sum of execution times. In order to create job execution times that all sum to a given value, the UUnifast algorithm as originally described by [1] is appropriate. In the UUnifast algorithm, $n-1$ execution times are sampled from a logarithmic distribution. The final value is then the difference between the sum of all previous values and the target value.

In the industrial case study it was observed that job execution times followed a logarithmic distribution, whereas task execution times followed a normal distribution. Yet in order to satisfy the job execution time distribution, the execution time of the tasks in a job must sum to a particular value. This distribution is created using a similar approach to UUnifast where $n - 1$ values are sampled, but from a normal instead of a logarithmic distribution. The last task execution time value is calculated, as before, to achieve the specified job execution time.

### 4.2 Stability

Stability can be measured by the percentage rate at which work is arriving into a grid compared to the maximum rate that the grid can process this work. The arrival rate is said to be stable if the arrival rate is less than the maximum processing rate (<100%), and unstable if the arrival rate is faster than the rate at which work can be processed (>100%) [3].

Grids are virtually always run at close to 100% stability ratio. Because the procurement and operational cost of a grid is very high, the operator is highly unlikely to over-buy resources for a grid. In addition, many computational loads can occupy as much computing power as is available. In many industrial grids, the stability ratio fluctuates around 100%. There may even be extended periods

**Algorithm 5** Pseudocode to define job arrival time with varying stability ratio

```
n = number of processors in system
jobcount = number of jobs in workload
sumj(i) = the sum of all task execution times in job i
p = desired stability ratio as percentage
start(i) = set the start time of job i

start(1) = 0
for j in 2 .. jobcount:
        singleprocworktime = sumj(j - 1) / n
        decimalp = p / 100
        start(j) = start(j-1) + (singleprocworktime / decimalp)
```

where the rate is over 100%, and the extra work must be queued. Therefore, it is necessary to be able to compare schedulers over a range of stability ratios.

A stability ratio for a workload can only ever be defined with relation to a platform, yet it is desirable to be able to adjust the stability ratio independently of the workload and platform. This can be achieved by adjusting the arrival times of jobs. The algorithm for calculating the arrival times of each job for a given platform and workload is given in Algorithm 5.

An alternative method for generating a workload is by specifying a duration of time for which the specified platform would be at 100% utilisation. The time slots on each processor are then divided up using the UUnifast-Discard algorithm, as presented by [6]. Each time slice on each processor then corresponds to a task. Each task is then randomly assigned a job to belong to. To vary the stability ratio for this method, the requested and actual finish time of the whole workload can be adjusted accordingly.

## 5 Conclusion

A summary of the issues surrounding Grid Scheduling were described, along with the background that motivates the work of this paper. Classes of Directed Acyclic Graph shapes and patterns that could be useful for evaluating the performance of schedulers were described, and algorithms for generating these patterns were shown. Algorithms for creating workloads with realistic distributions of task and job execution times were presented. The issue of stability was described, as were two methods of creating workloads with a given stability level for given platforms.

Several areas of future extensions to this work are possible. There may be more possible patterns of DAG that could be deduced from further case studies. Other workloads may demonstrate distributions of job and task execution times other than logarithmic and normal, respectively.

The performance of the implementation of these algorithms should be evaluated. The algorithms presented here for generating workloads can struggle with a high rate of discarding when certain parameters are set to extremes. Investigation into algorithms that still produce tasks and jobs with the desired distributions but eliminate or minimise the discard rate would also be valuable to increase the efficiency of workload generation. From a scheduling aspect, determining which schedulers are best suited to working with each kind of task graph shape is an ongoing topic of research.

The work presented in this paper is intended to demonstrate the algorithms used in the creation of an application model. This application model, when combined with a future platform model, will be used to compare different scheduling policies as to their effectiveness on a variety of grid workloads.

## References

1. E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-time Systems*, 30:129–154, 2005.
2. P. Brucker. *Scheduling Algorithms.* SpringerVerlag, 2004. ISBN 3540205241.
3. S. J. Chapin. Distributed and multiprocessor scheduling. *ACM Comput. Surv.*, 28(1):233–235, 1996. ISSN 0360-0300.
4. D. E. Collins and A. D. George. Parallel and sequential job scheduling in heterogeneous clusters: A simulation study using software in the loop. *Simulation*, 77(5-6):169–184, 2001.
5. B. Dally. Life after moore's law. Forbes.com, April 2010.
6. R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems Symposium, IEEE International*, 0:398–409, 2009. ISSN 1052-8725.
7. P. Emberson. *Searching For Flexible Solutions To Task Allocation Problems.* Ph.D. thesis, University of York, UK, 2009.
8. P. Emberson, R. Stafford, et al. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 6–11. Jul. 2010.
9. M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.
10. R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.
11. C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Nov. 1998. ISBN 1558604758.
12. M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, p. 57. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8365-1.
13. R. Miller. Special report: The world's largest data centers. April 2010.
14. G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965.
15. V. Salapura. Next generation supercomputers. IBM.com, October 2007.
16. A. Schoneveld, J. F. de Ronde, et al. On the complexity of task allocation. *Complex.*, 3(2):52–60, 1997. ISSN 1076-2787.
17. S. Selvakumar and C. S. R. Murthy. A list scheduling anomaly. *Microprocessors and Microsystems*, 17(8):471 – 474, 1993. ISSN 0141-9331.