Group-Based Parallel Multi-Scheduler for Grid Computing

Goodhead T. Abraham, Anne James and Norlaily Yaacob

Distributed Systems and Modelling Research Group,
Coventry University, UK, CV1 5FB
abrahamg@uni.coventry.ac.uk
{a.james, n.yaacob}@coventry.ac.uk

Abstract

With the advent in multicore computers, the scheduling of Grid jobs can be made more effective if scaled to fully utilize the underlying hardware, and parallelized to benefit from the exploitation of multicores. The fact that sequential algorithms do not scale with multicore systems nor benefit from parallelism remains a major obstacle to scheduling in the Grid. As multicore systems become ever more pervasive in our computing lives, over reliance on such systems for passive parallelism does not offer the best option in harnessing the benefits of their multiprocessors for Grid scheduling. An explicit means of exploiting parallelism for Grid scheduling is required. The Group-based Parallel Multi-scheduler, introduced in this paper, is aimed at effectively exploiting the benefits of multicore systems for Grid scheduling by splitting jobs and machines into paired groups and independently scheduling jobs in parallel from those groups. We implemented two job grouping methods, *Execution Time Balanced (ETB)* and *Execution Time Sorted then Balanced (ETSB)*, and two machine grouping methods, *Evenly Distributed (EvenDist)* and *Similar Together (SimTog).* For each method, we varied the number of groups between 2, 4 and 8. We then executed the MinMin Grid scheduling algorithm independently within the groups. We demonstrated that by sharing jobs and machines into groups before scheduling, the computation time for the scheduling process drastically improved by magnitudes of 85% over the ordinary MinMin algorithm when implemented on a HPC system. We also found that our balanced group based approach achieved better results than our previous *Priority* based grouping approach.

Keywords: Grid Scheduling, Multicore systems, Parallelism, Multi-scheduling, Machine Grouping, Job Grouping, HPC.

1. Introduction

With the advent of multicores, scheduling of Grid jobs can be made more effective if parallelized to fully utilize the multicore and benefit from the underlying hardware. Most Grid scheduling algorithms are saddled with overheads incurred in the pre-optimizing computations done before scheduling jobs. Secondly, during scheduling, more overheads are incurred when new jobs arrive and the whole pre-optimizing computations have to be done over again. Furthermore serial scheduling algorithms become bottlenecks when the number of tasks to be scheduled grows.

Multicore technology has come to stay and as Grid computing continues to grow, it will be worthwhile to scale Grid scheduling to benefit from the multicore technology. Multicore systems offer opportunity for parallelism and increased throughput. Parallelism takes programming away from the traditional serial execution approach by employing several

processors to simultaneously execute independent tasks and is best suited for independent jobs which characterize a large percentage of users' jobs on the Grid. Increased throughput is a direct increase in output over a set period resulting from more efficient processing. Current Grid scheduling algorithms do not exploit the benefits inherent in the underlying multicore systems, mostly focussing on parallel execution of jobs rather parallelising the scheduling function. Neglecting the underlying multicore hardware in the scheduling algorithm of the Grid will cause an unnecessary bottleneck in processing.

The design of a parallel multi-scheduling method for the Grid that takes the underlying multicore hardware into consideration will help position the growth on the right path for future challenges. This work is aimed at exploiting the benefits of multicore systems for the improvement of Grid scheduling. This research builds on our previous work, the Priority-based Parallel Multi-scheduler (PPMS) method [1]. We found in our previous work that grouping tasks and machines and then scheduling in parallel across paired groups can improve scheduling time. The savings occurred because of the reduction in job numbers in the groups which dampened the polynomial shape of the scheduling algorithm, MinMin, and also because of the parallelisation. We also noticed that the heuristics employed in configuring the groups effects the scheduling time. The work in this paper sets out a generic grouping approach that can be calibrated according to various criteria such that improved performance can be obtained. It generalises our previous work and offers further discussion on its potential impact.

We used two methods for grouping of Grid jobs, and two methods for the grouping of Grid resources (machines). After the grouping of machines and jobs separately, a pairing is made between job groups and machines groups. Then using multiple threads (multi-threading), scheduling is executed independently within the paired groups in parallel. Multi-scheduling in this context refers to the scheduling of several independent groups of jobs to groups of machines in parallel. A resources group contains a set of different computers for servicing a set of jobs from a job group – the machines are grouped based on their configuration. Two methods are adopted for this purpose: and *Evenly Distributed (EvenDist);* and *Similar Together (SimTog)*. The methods are discussed in section 3. A job group contains a set of Grid jobs submitted by users but sorted into a group, based on some characteristics for the purpose of being scheduled to a machine group independently. Two methods are adopted for categorizing jobs into groups: *Execution Time Balanced (ETB);* and *Execution Time Sorted and Balanced (ETSB)*. These methods are also discussed in section 3. The scheduling of jobs takes place within the groups simultaneously, achieved through the use of threads.

The remainder of the paper is organized as follows; Section 2 discusses related work. Section 3 presents the Group-based Parallel Multi-Scheduling (GPMS) method. Section 4 describes the simulation for the experiment and section 5 discusses the experimental setup and scenarios. Section 6 discusses results, analysis of the results and performance evaluation of the method against the MinMin and also against the PPMS method. Section 7 provides further discussion of results. Section 8 provides conclusion and thoughts for future work.

2. Related Work

The scheduling problem in heterogeneous environments is NP-complete [2]. Typically heuristics are employed to ease the problem solving. A parallel scheduler for the Grid would reduce further the time needed to solve the scheduling problem. Such a design will exploit multicore technology and play a major role in the defined growth path of the Grid, facilitating increased throughput and scalability. The challenge is to develop a parallel scheduler that is dynamic, adaptive to workload increase, ensures increased throughput, is free of bottlenecks and which maximizes resource utilization.

Scheduling can be carried out in immediate mode or batch mode [3]. Immediate mode is when a job is assigned to a machine as it arrives and batch mode is when a number of jobs are batched and scheduled together. Algorithms for immediate mode include: the traditional First Come First Serve (FCFS); Backfilling Opportunistic Load Balancing (OLB); minimum execution time (MET); minimum completion time (MCT); and k-percent best (KPB). Some examples that use these approaches are [4, 5, 6, and 7]. Quezada-Pina et al. [8] compared some of these strategies in the context of a strategy of admissible machines, where only part of the complete set of machines is made available. Their simulation results revealed that in terms of the considered criteria, admissible allocation strategies outperform algorithms that use all available sites for job allocation. Liang et al. [9] used behavioural clustering of execution time to establish a pattern for users' jobs and used that to improve accuracy of overall job execution times. Batch approaches include algorithms such as: MinMin, where jobs with the minimum completion time are assigned to the processor that can complete the job the earliest; MaxMin, where jobs with maximum completion time are assigned to processors that can complete the job earliest; and Sufferage, where a machine is assigned to the task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it [3]. Evolutionary models have also been applied [10, 11] in a batch context. Previous work has investigated and compared different immediate and batch mode scheduling algorithms [3, 8]. Further discussion is provided in our previous work [1].

Later work has taken more of a user perspective and concentrated on providing or maintaining Quality of Service. These efforts include: resource reservation mechanism [12], monitoring, reallocation, varying levels of service [13]; and deadline guarantees [10, 14]. All the schemes mentioned in this paragraph concentrate on makespan (the time taken to execute a set of jobs) or improving quality of service to users in terms of job execution time and cost. They do not concentrate on improving the efficiency of the scheduler in terms of how long the scheduling task takes. Our current research aims to improve the efficiency of the scheduler through exploiting parallelisation in a new way. This will also improve makespan and quality of service further.

There has been some work that has addressed parallelisation of the scheduler itself [1]. Nesmachnow and Canabé [15, 16] investigated the use of massively parallel GPUs (Graphical Processing Units) to improve scheduling time. Pinel, Dorronsoro and Boury [17] have presented CPU and GPU multi-threaded parallel designs of the MinMin algorithm. As would be expected, the GPU design outperforms the CPU because of the massive parallelisation, while the parallel CPU solution outperformed the serial. The experimental evaluation of the proposed parallel methods demonstrates that a significant reduction on the computing times can be obtained when using the parallel GPU hardware. Further approaches have proposed evolutionary algorithms which exploit GPUs in solving the scheduling problem [18, 19]. Nesmachnow, Cancela and Alba [20] implemented a parallel micro evolutionary algorithm to schedule tasks in heterogeneous and Grid environment algorithm. Speed-up was obtained in comparison to their previous work. Pinel, Dorronsoro and Boury [17] proposed a cellular genetic algorithm (CGA) to solve the MinMin problem. The CGA brought better solutions but took longer to run. Mirsoleimani, Karami and Khunjush [21] propose a memetic algorithm, which uses combinations of non-deterministic approaches to solve the scheduling problem in a GPU environment. Very high speed-up was achieved. The difference between the research described in this paragraph and our research is that the other research described has focussed mainly on a GPU environment and/or on non-deterministic algorithms such as evolutionary algorithms. The GPU environment offers massive parallelisation. However the non-deterministic algorithms can have unpredictable run times. The scope of our work has been the more general purpose environment. We selected this environment as we wanted to create a facility that did not require a specialised environment. We also concentrated on deterministic algorithms to have better control on scheduler

execution time. The other novelty in our work is the use of grouping to achieve greater efficiency through parallelisation.

In our previous work on Priority-based Parallel Multi-scheduler (PPMS) for the Grid, we noted that the effect of grouping machines and jobs before scheduling was promising but the group cardinality (i.e. number of groups) we used was constant [1]. We needed to investigate further by employing methods that can vary the number of groups. In that same work, we equally noted that the *Priority* method does not necessarily distribute jobs equally among the job groups. This phenomenon degraded the general performance. We needed to develop a method that equitably distributes jobs amongst the groups which we have called Group-based Parallel Multi-scheduler (GPMS) for the Grid. The method employs various grouping strategies to exploit the multicore. It offers a platform of parallelism to the scheduling algorithm itself to improve the computation and scheduling times and also provide answers to some of the questions thrown up in our earlier work.

To summarise, parallel multi-scheduling can improve Grid scheduling performance and should be exploited. This research aims to exploit multicores both on scheduler and on Grid sites through an innovative grouping method which enhances and optimizes the performance of Grid schedulers.

3. Group-Based Parallel Multi-Scheduler (GPMS) for Grid

3.1 Overview of the scheduler

In our previous work on Priority-based Parallel Multi-scheduler (PPMS) for the Grid [1], we focused on the use of grouping of Grid jobs based on their priorities and categorizing Grid machines based on two methods, the *Evenly Distributed* and *Similar Together* methods. We noted it will be worthwhile to exploit further the effects of varying the number of groups for both jobs and machines. We also needed a method that distributes jobs equally among the job groups as the *Priority* method does not guarantee this.

The Group-based Parallel Multi-scheduler (GPMS) for Grid aims at exploring parallelism on multicore systems to enhance scheduling algorithms in Grid. To achieve this we assume that multicores are pervasive and constitute major part of Grid machines. We also assume that our scheduler runs on a multicore system.

The GPMS requires jobs to be split into groups. We used two methods to achieve this:

Execution Time Balanced (ETB) - Estimate execution time and then balance across groups.

Execution Time Sorted and Balanced (ETSB) - Estimate execution time, then sort jobs and then balance across groups.

Jobs are first read into the scheduler by a job reader. We then estimate the execution time for each job. The resulting job size and estimated execution time statistics are held in a table (the Estimation table) to be used later for grouping and/or scheduling decisions.

The GPMS also requires machines to be split into the same number of groups as the jobs. We use two methods to achieve this:

Evenly Distributed (EvenDist) - machines are evenly distributed independent of characteristics.

Similar Together (SimTog) - machines with similar characteristics are grouped together.

Information about Grid machines such as machine id., CPU speed and number of CPUs are known to the algorithm and are used for splitting decisions and also for simulation and computation of execution times of jobs. The scheduling algorithm is then executed in parallel within the groups. A thread pool is created to enable the parallel execution within the groups (multi-threading). The scheduling of jobs takes place simultaneously and independently between paired groups (multi-scheduling). The MinMin algorithm is used within each group pair to schedule the jobs.

The steps of the GPMS are presented in Table 1. Figure 1 provides an illustrative activity diagram showing the GPMS steps of two processes. One is the machine grouping process and the other the process of grouping and scheduling jobs onto machines. The machine grouping process occurs less frequently that the scheduling process.

Table 1: High-level algorithm for GPMS

Step1.	Start
Step2:	Specify number of threads
Step3:	Specify number of groups to use
Step4:	Read jobs into the scheduler
Step5:	From the job attributes, estimate the execution time for each job
Step6:	Group jobs into number of specified groups using a chosen grouping method
Step7:	Read machines and group them into the specified number of groups using a chosen grouping method
Step8:	Execute the scheduling algorithms within the groups
Step9:	Write results to output file
Step10:	Stop

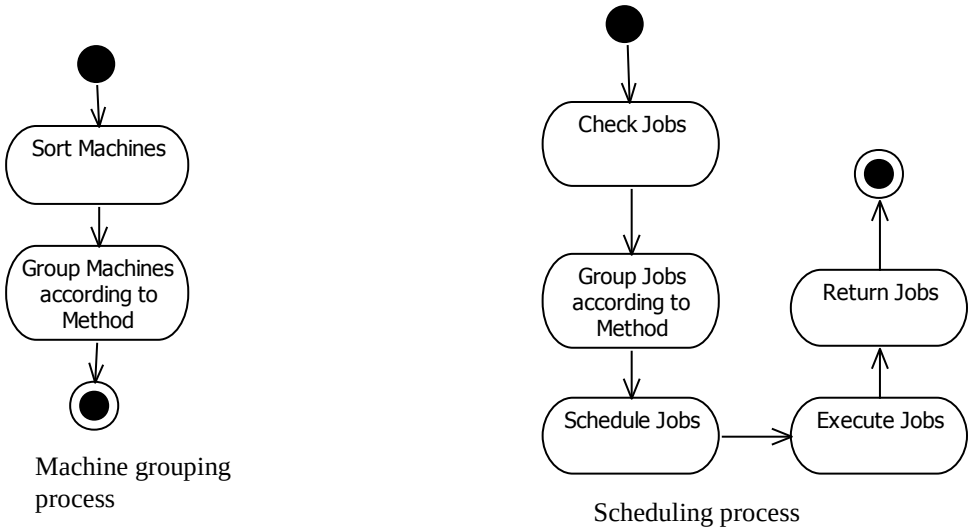

Figure 1: Activity diagram showing steps of the GPMS

3.2 Overview of grouping methods

The GPMS requires jobs and machines to be grouped. The groups are paired and scheduling occurs in parallel within the groups. Various methods could be used to group the jobs and machines. In this paper we discuss four methods, two for job grouping and two for machine grouping.

3.2.1 Grouping jobs

Our approach splits jobs into groups before executing the scheduling algorithm within the groups. Jobs are split (grouped) based on the estimated execution time computed from their size. Jobs are initially held in a table which also holds their estimated size. Two methods are employed in splitting jobs into groups:

Execution Time Balanced (ETB)
This method uses an estimation of the processing time for each job to group the jobs. It attempts to even out the total processing times in groups by adding the next job to the group with the current lowest total processing time. Table 2 shows the algorithm for the ETB method.

Table 2: Algorithm for the ETB method of grouping jobs

Step1:	Start
Step2:	Select job from the Estimation table
Step3:	Select the group with the smallest **totalestimatedTime**
Step4:	Add job to group with the smallest **totalestimatedTime**
Step5:	Update the **totalestimatedTime** for the group
Step6:	Repeat step2 to step5 until end of table
Step7:	Stop

Execution Time Sorted and Balanced (ETSB)

This method is similar to the ETB method but this time the jobs are sorted by size first, with the largest jobs at the top of the list. This has the effect of ensuring a fairer balance across groups. Table 3 outlines the ETSB algorithm.

Table 3: Algorithm for the ETSB method of grouping jobs

Step1:	Start
Step2:	Sort jobs in the Estimation table
Step3:	Select job from the Estimation table
Step4:	Select the group with the smallest **totalestimatedTime**
Step5:	Add job to group with the smallest **totalestimatedTime**
Step6:	Update the **totalestimatedTime** for the group
Step7:	Repeat step3 to step6 till end of table
Step8:	Stop

3.2.2 Grouping machines

For every group of jobs, there is equally a group of machines onto which jobs are to be scheduled. This is to allow a high degree of independence of job and machine groups and to enable scheduling to take place in parallel over the groups. Two methods are employed in grouping (splitting) machines:

Evenly Distributed (EvenDist)

This method splits machines into each group so that each group has the same number of machines. It adds the first machine to the first group, then the second to the second group, third to the third group and so on until the last group is reached. When the last group is reached the next machine is allocated to the first group and the process continues until all machines are allocated. Machines are first sorted before the grouping. This method ensures that machines making up the Grid are equally (or best effort equally) split into the specified groups. This method offers better result if jobs are equally distributed among groups and if the scheduling policy does not favour one set of jobs over another. Table 4 shows the algorithm for the Evenly Distributed method.

Table 4: Algorithm the Evenly Distributed method for grouping machines

Step1: Start
Step2: Sort machines based on configurations (i.e. number and speed of processors)
Step3: Register number of groups
Step4: Add first machine to first group
Step5: Add next machine to next group
Step6: Repeat Step4 and Step5 until last group is reached.
Step7: Add next machine to first group
Step8: Repeat Step4, Step5 and Step6 until all machines are assigned to groups.
Step9: Stop

Similar Together (SimTog)

This method groups machines based on their performance characteristics. To group machines based on their similarity, the configurations of the machines are compared and those with similar characteristics (i.e. CPU speed and number of CPUs) are grouped together before jobs are scheduled to them. This type of grouping can be useful if a priority-based scheduling method is to be used and the jobs are similarly distributed in terms of priority. Each group has the same number of machines. Table 5 shows the algorithm for the Similar Together method.

Table 5: Algorithm for the Similar Together method for grouping machines

Step1: Start
Step2: Sort machines based on configurations (i.e. number and speed of processors)
Step3: Work out how many machines per group (N = number machines/number groups)
Step 4: Add top N machines to the first group,
Step5: Add next N machines to the next group
Step6: Repeat Step5 until all machines are assigned
Step7: Stop

3.2.3 Inside group scheduling

Our grouping method seeks to improve the efficiency of Grid scheduling algorithms by enabling the parallel multi-scheduling of jobs between independent groups of machines and jobs. After grouping both jobs and machines with our method, we then implemented the traditional Grid scheduling algorithm inside the groups. In this study, we implemented the MinMin scheduling algorithm within the groups.

3.2.4 Multi-threading

Multi-threading was implemented with a dynamic thread pool. Threads were activated when needed and deactivated when no longer needed. With the thread pool, we had the option of

choosing in our test parameters how many threads to use for each execution. Threads are not necessarily set in a one to one relationship with each parallel scheduling event. In this paper we have analysed the results for 4 threads.

4. The Simulation

Simulation was used for the experiment. In this section we discuss the source for the jobs used in the experiment, how the Grid was simulated, and how the execution time was estimated.

4.1 Grid job source

Jobs used for our experiment were downloaded from the Grid Workload Archive [22]. The Grid Workload Archive is designed to make traces of Grid workloads available to researchers and developers alike. It contains files both in plain text format and the Grid Workload Format (GWF). The GWF file contains 29 attributes concerning the running of a job in a Grid. We used some of these to create an estimate of job size and execution time. Of the attributes contained in the GWF file, we employed only the fields relevant for our simulation, estimation and experiment. These attributes are used to estimate job execution times on the Grid machines. Table 6 shows relevant job attributes relevant in our work.

Table 6: Attributes from the Grid Work Archive used in our work

Attribute	Description
JobID	Identifies the job
NProcs	Number of allocated processors
ReqTime	Requested time measure in wallclock seconds
ReqNProcs	Requested number of processors
RunTime	Measured in wallclock seconds
AverageCPUTime	Average CPU time over all the allocated processors

4.2 Estimation of Job Sizes

The information in the GWF trace file does not consist of job sizes but our simulation and experiment needs this variable to estimate the execution time of the job on a processor. We therefore estimate the job size from some available attributes, such as those shown in Table 6. For instance if we multiple RunTime by NProcs we have some estimate of size. A more accurate estimate may come from AverageCPUTime multiplied by NProcs but AverageCPUTime was not always available. A value not available is shown as -1 in the file. Table 7 shows typical values of the attributes from some of the rows in the trace file. Because of missing values it was not possible to accurately replicate original job size from the trace file but we used some values available to generate a set of jobs with estimated sizes. The estimations served our experiment adequately as we needed a range of jobs of varying sizes with which to experiment. Exactly mirroring the sizes of the jobs in the trace file was not necessary.

Table 7: Typical values from the trace file

JobID	RunTime	NProcs	AverageCPUTime	ReqNProcs	ReqTime
0	0	4	-1	4	3600
1	19	1	-1	1	3600
2	10	5	-1	5	3600
3	8	90	-1	90	3600
4	19	100	-1	100	3600
5	25	1	-1	1	3600

4.3 Simulation of Grid

A Grid site was characterized by the following attributes: Category; CPU; RAM; Bandwidth. For example {A; 1200; 1000} represents Grid site A, with 1200 CPUs and Bandwidth 1000 Mbps. A Machine is defined with the following attributes: CORES; CPU; RAM. For instance {2; 2000; 2000000} represents a Grid resource (machine) with 2 CPUs, 2000 MHz (2GHz) and 2000000 (2MB) RAM. Table 8 shows the characteristics of the simulated Grid.

Table 8: Components of the simulated grid

Grid Site	Characteristics			Grid Site	Characteristics		
	Number of machines	Speed of CPU	Number of CPU/Cores		Number of machines	Speed of CPU	Number of CPU/Cores
A 240 Machines	30 30 30 30 30 30 30 30	1GHz 2GHz 3GHz 4GHz 1GHz 2GHz 3GHz 4GHz	1 1 1 1 2 2 2 2	C 440 Machines	60 60 60 60 60 60 60 60	1.5GHz 2GHz 3.5GHz 4GHz 1.5GHz 2GHz 3.5GHz 4GHz	2 2 2 2 4 4 4 4
B 400 Machines	50 50 50 50 50 50 50 50	1.5GHz 2GHz 3.5GHz 4GHz 1.5GHz 2GHz 3.5GHz 4GHz	2 2 2 2 4 4 4 4	D 600 Machines	50 50 50 50 50 50 50 50 50 50 50 50	1.5GHz 2GHz 3.5GHz 4GHz 1.5GHz 2GHz 3.5GHz 4GHz 1.5GHz 2GHz 3.5GHz 4GHz	2 2 2 2 4 4 4 4 8 8 8 8

4.4 Simulation of execution time

The execution time was estimated from the job sizes. The processing speeds of Grid machines were used to estimate execution times. Waiting times were calculated according to the schedule and allocation to individual machines. Table 9 gives an outline algorithm for calculating execution time.

Table 9: Simulation of execution time

Step 1: Set the job size to be job execution time (T) on a reference machine (1 GHz, 1core)

Step 2: Scale the expected time to match the current machine

a. Calculate performance ratio (R) between the current and the reference machine
b. Return the expected execution time divided by the performance ratio (T/R)

5. Experimental Design

We carried out our experiments on a HPC system using one node. In the experiment, we simulated a Grid environment with four Grid sites consisting of machines with different CPU

speed and number of processors. Our scheduling aims directly at the CPUs on the individual machines. Jobs are scheduled to CPUs.

The configuration of the HPC machine node on which our experiment was executed is as follows:

Number of physical CPUs per node/head: 2
Numbers of cores per one compute node/head: 12
CPU family: Intel(R) Xeon(R) CPU X5650 2.67 GHz stepping 02
Operating System: Linux x86_64 RHEL 5

For each of the experiments described below we calculated for each parameter variation, the time taken to schedule and the flow time. The flow time is the time taken to execute all of the jobs. We defined makespan as the time to schedule plus the flow time. The scheduling time is the time taken to schedule all the jobs using the algorithm.

Experiment 1

In the first experiment, we executed the MinMin algorithm on the HPC to schedule a range of jobs (from 1000 jobs to 10000 jobs in steps of 1000) using 4 threads. In each instance of the experiment, we recorded the time taken to schedule each step of jobs by the set number of threads, for instance the time taken to schedule 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10,000 jobs. We also calculated the makespan for each variation.

Experiment 2

In the second experiment, we used the ***ETB*** method to group the jobs and the ***EvenDist*** method to group the machines before implementing the same test as described in Experiment 1 above. We used 2, 4 and 8 groups and 4 threads in each case. We recorded time taken to schedule and the makespan each variation.

Experiment 3

In the third experiment, we used the ***ETB*** method to group the jobs and the ***SimTog*** method to group the machines before implementing the same test as described in Experiment 1 above. We used 2, 4 and 8 groups and 4 threads in each case. We recorded time taken to schedule and the makespan for each variation.

Experiment 4

In the fourth experiment, we used the ***ETSB*** method to group the jobs and the ***EvenDist*** method to group the machines before carrying out the same test described in Experiment 1 above. We used 2, 4 and 8 and 16 groups and 4 threads in each case. We recorded time taken to schedule and the makespan for each variation.

Experiment 5

In the fifth experiment, we used the ***ETSB*** method to group the jobs and the ***SimTog*** method to group the machines before implementing the same test described in Experiment 1 above. We used 2, 4, 8 and 16 groups and 4 threads in each case. We recorded time taken to schedule and the makespan for each variation.

Our interest in this research was on improvements on scheduling in terms of throughput – the gains we can make in improving the number of jobs scheduled over a period of time. The aim was to ameliorate the issue of scheduler bottleneck through scheduling in parallel.

6. Performance Evaluation

In this section, we present results, analysis and performance of our GPMS method against results from the MinMin algorithm used in isolation. We shall refer to results from Experiment 1 (the first experiment done without the grouping method) as ordinary MinMin or MinMin, while we shall properly label our results for the other experiments with a combination of both job and machine grouping methods and the number of groups. We employed linear graphs, bar charts, tables, trend lines, mathematical computations, percentage computation, correlation and standard deviation and ANOVA statistical significance methods for the analysis and evaluation of our results against the MinMin results. We also compared our results against the results of our previous work which used the *Priority* method for grouping jobs [1].

Our complete experimentation yielded many results because of the combinations of several variables (number of groups, number of threads, job grouping method, machine grouping methods). The analysis in this paper is based on results for each group cardinality (i.e. 2Group, 4Group and 8Group excluding 16 groups) and for each method (i.e. MinMin, *ETB-EvenDist*, *ETB-SimTog*, *ETSB-EvenDist*, and *ETSB-SimTog*). We kept the thread number to 4 for this analysis so that we could focus on improvement, independent of thread variability. Four threads was chosen because it represents the median cardinality of threads used (1, 2, 4, 8, 16) in our complete experimentation. Also, it allows us to compare results against the *Priority* (PPMS) method from our previous work which used 4 threads.

In our performance evaluation, we investigate the following factors and their effects:

- Group cardinality (section 6.1).
- Varying grouping methods (section 6.2).
- GPMS method compared to our previous *Priority* (PPMS) method (section 6.3).

6.1 Group Cardinality

In this section we analyse the effect of group cardinality (number of groups) using the following methods:

1. MinMin versus ETB and Evenly Distributed method (*ETB-EvenDist*)
2. MinMin versus ETSB and Similar Together method (*ETSB-SimTog*)

For the sake of brevity we do not include all four method combinations in our analysis of group cardinality. All methods showed a similar pattern, with *ETB-EvenDist* an example of a higher performing method over MinMin and *ETSB-SimTog* showing the lowest performance over MinMin. The selection of the above methods for analysis therefore gives an illustration of the range of performance improvement over MinMin and the range of methods. Note that three of the methods demonstrated equally high performance with no statistically significant difference between them. These were *ETB-EvenDist*, *ETB-SimTog* and *ETSB-EvenDist*. Thus any of those three would be an example of the highest performance. We see later that *ETSB-SimTog* was the odd one out with a different behaviour. This was the method that gave the lowest performance, although still significantly better than MinMin alone.

6.1.1 MinMin vs. ETB and Evenly Distributed (ETB-EvenDist) method

Table 10 shows the results and computation of improvement of the GPMS *ETB-EvenDist* method over the MinMin. These results were yielded after running experiment 1 and experiment 2 (see section 5).

The MinMin method used a total of 242033 and an average of 3486.2 ms to schedule the job sets. Using two groups, our method used a total of 34862 and an average of 3486.2 ms to schedule same task. Four groups used a total of 4701 and an average of 470.1 ms to schedule the task, while eight groups used a total of 1435 and an average of 143.5 ms to schedule same tasks. Figure 2 shows the results graphically and Table 11 provides the ANOVA test results which reveal the significance of the differences between the groups. Taking P values less than 0.05 to indicate significance, we see that all differences were found to be highly significant with very low P values.

Using two groups, the *ETB-EvenDist* method recorded between 6.32 and 9.92 speed-up with an average of 7.62 speed-up against the MinMin. This represents an average of 86.53 percent performance improvement over the MinMin. Using four groups, the method recorded between 16.35 to 59.19 speeds up with an average of 47.46 speed-up over the MinMin, representing 97.58 percent improvement over the MinMin. Eight groups recorded between 59.45 and 182.50 speed-up and an average of 155.33 speed-up over the MinMin, representing 99.28 percent performance improvement over the MinMin. Across all the groups, as the number of jobs increases, there was a general improvement in the speed-up to a point beyond which the speed-up declines.

There was a significant performance improvement by the GPMS over MinMin as the number of groups increased. Increasing the number of groups enabled the scheduler to compute and schedule more jobs in parallel. Figures 3 and 4 illustrate the scale of performance improvements in multiples and percentage against MinMin. We can see an improvement of scheduling efficiency with respect to the increase in groups. As the scheduling changes from two groups to eight groups, the scheduling efficiency improved. This shows that using more groups increases the performance of the scheduling algorithm. Figure 5 shows the improvement in scheduling time as the number of groups increase. However the rate of performance improvement of a successive group over its predecessor decreases generally, for instance, using the *ETB-EvenDist* method, the rate of improvement of two groups over the ordinary MinMin was 6.94, between 2 groups and 4 groups, there was performance improvement of 7.41 while between 8 groups and 4 groups, performance improved only 3.28 times. This shows that even though there is a general performance improvement over MinMin with increasing groups, the rate of improvement declines and levels off as the number of groups increases. The calculation of this decreasing rate of improvement is given in Table 12 and illustrated by a line graph in Figure 6.

Table 10: Scheduling times and speed-up for MinMin vs. ETB-EvenDist

Methods	**MinMin Vs. ETB-EvenDist Time in ms**				**Speed-up (X) in multiples**			**Speed-up (%) in percentage**		
Jobs Limit	**MinMin**	**2Grps**	**4Grps**	**8Grps**	**2Grps**	**4Grps**	**8Grps**	**2Grps**	**4Grps**	**8Grps**
1000	654	101	40	11	6.48	16.35	59.45	84.56	93.88	98.32
2000	3230	331	92	25	9.76	35.11	129.20	89.75	97.15	99.23
3000	7601	766	163	46	9.92	46.63	165.24	89.92	97.86	99.39
4000	12920	1475	252	76	8.76	51.27	170.00	88.58	98.05	99.41
5000	18219	2410	323	100	7.56	56.41	182.19	86.77	98.23	99.45
6000	22671	3211	383	128	7.06	59.19	177.12	85.84	98.31	99.44

7000	29504	4670	511	185	6.32	57.74	159.48	84.17	98.27	99.37
8000	39074	5565	729	228	7.02	53.60	171.38	85.76	98.13	99.42
9000	48178	6989	954	294	6.89	50.50	163.87	85.49	98.02	99.39
10000	59982	9344	1254	342	6.42	47.83	175.39	84.42	97.91	99.43
Total	**242033**	**34862**	**4701**	**1435**	**76.19**	**474.63**	**1553.32**	**865.27**	**975.80**	**992.84**
Average	**24203.3**	**3486.2**	**470.1**	**143.5**	**7.62**	**47.46**	**155.33**	**86.53**	**97.58**	**99.28**

Table 11: ANOVA results for ETB –EvenDist method vs. MinMin and between groups

Test	Method	P-value	Significant Difference?
1	MinMin / ETB-EvenDist (All groups)	0.001995	Yes
2	MinMin/ ETB-EvenDist (2Grps)	0.00431	Yes
3	MinMin/ ETB-EvenDist (4Grps)	0.00136	Yes
4	MinMin/ ETB-EvenDist (8Grps)	0.00121	Yes
5	ETB-EvenDist (2Grps)/ ETB-EvenDist (4Grps)	0.006842	Yes
6	ETB-EvenDist (2Grps)/ ETB-EvenDist (8Grps)	0.003126	Yes
7	ETB-EvenDist (4Grps)/ ETB-EvenDist (8Grps)	0.022274	Yes

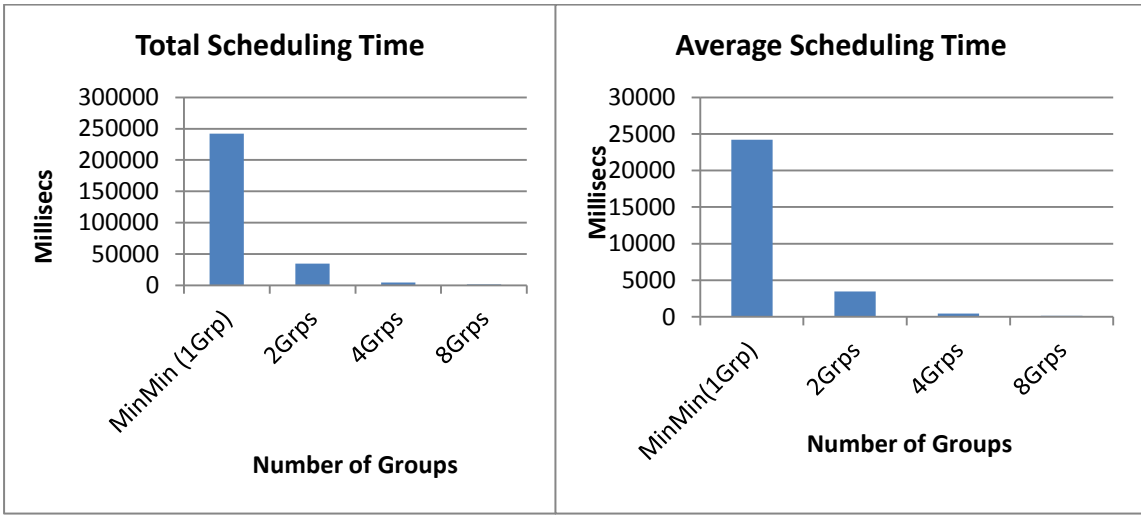

Figure 2: Total and average scheduling times across groups (ETB-EvenDist)

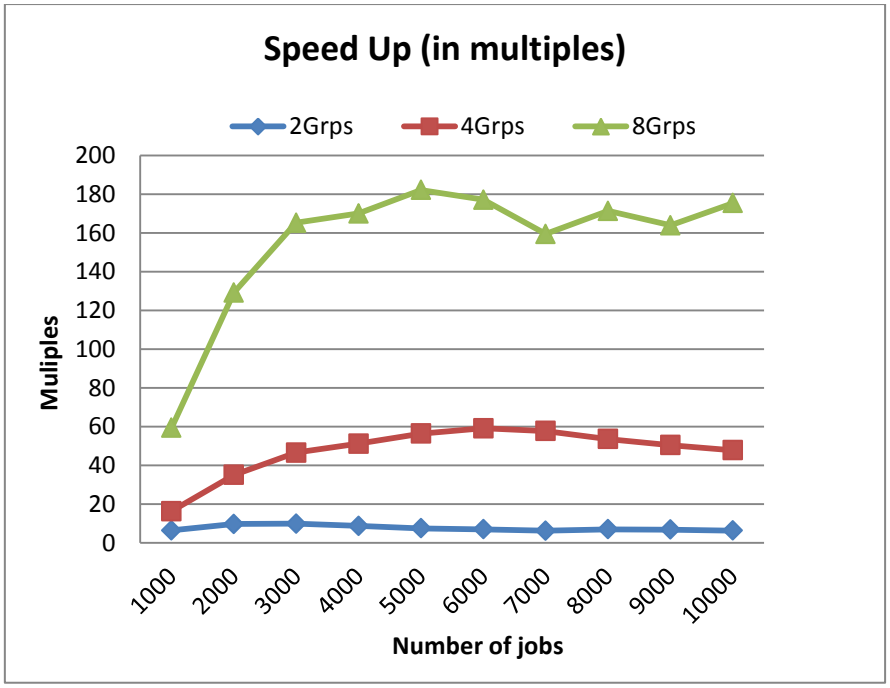

Figure 3: Speed-up (in multiples) of the ETB_EvenDist over MinMin with increasing jobs

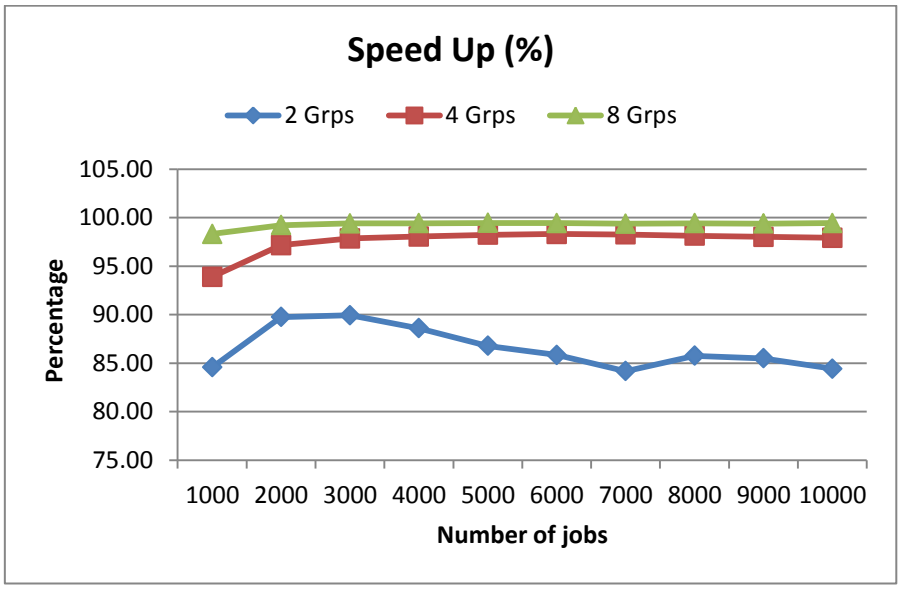

Figure 4: Speed-up (in percentage) of the ETB-EvenDist over the MinMin with increasing jobs

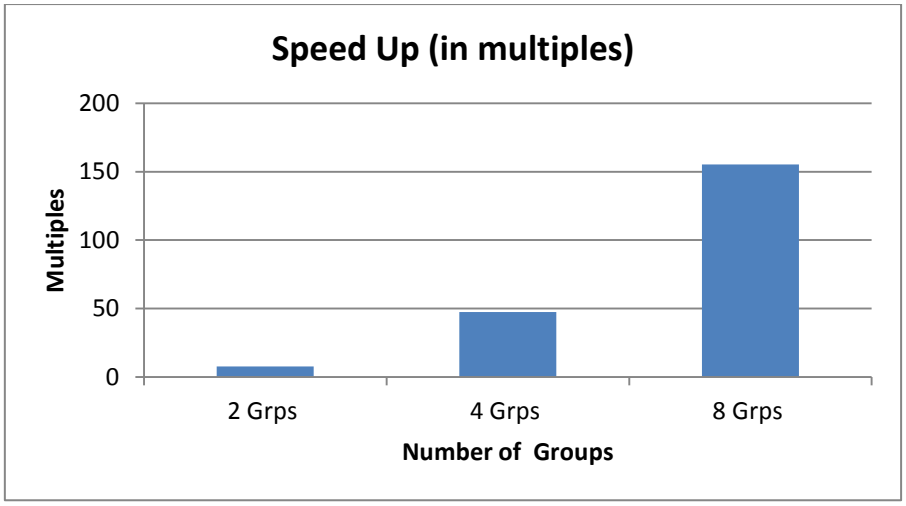

Figure 5: Improvement of the ETB-EvenDist method across group cardinality

Table 1 Performance improvement for ETB-EvenDist method vs. MinMin and between groups

Methods		**ETB-EvenDist Performance Improvement(X)**			**ETB-EvenDist Performance Improvement (%)**			
Algorithm	**MinMin**	**2Grps**	**4Grps**	**8Grps**	**Methods**	**2Grps**	**4Grps**	**8Grps**
Total	242033	34862	4701	1435				
$\frac{Total_{MinMin}}{Total_{Group}}$		6.94	51.49	168.66	$\frac{x_1 - x_2}{x_1} X100$ x_1 = MinMin	85.60 x_2 =2Grps	98.06 x_2 =4Grps	99.41 x_2 =8Grps
$\frac{Total_{nGrps}}{Total_{n+1Grps}}$ $n \in [2,4,8]$			7.41	24.29	x_1 = 2Grps		86.52 x_2 =4Grps	95.88 x_2 =8Grps
$\frac{Total_{nGrps}}{Total_{n+1Grps}}$ $n \in [2,4,8]$				3.28	x_1 = 4Grps			69.47 x_2 =8Grps

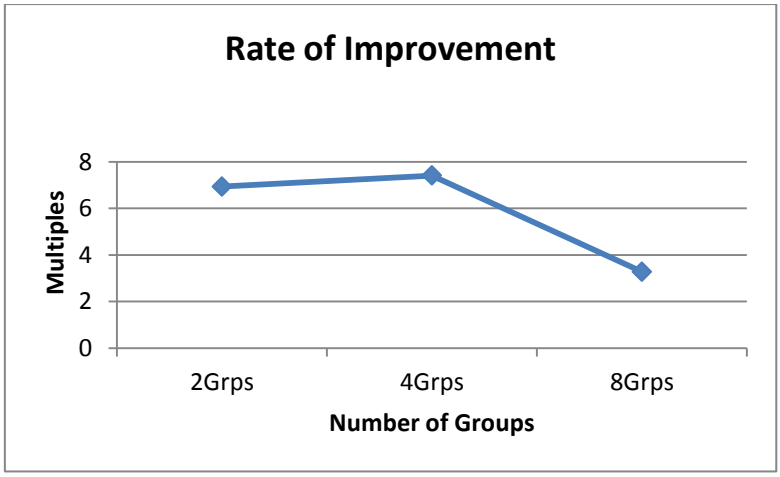

Figure 6: Speed-up rate as group cardinality increases (ETB –EvenDist)

6.1.2 MinMin vs. ETSB and Similar Together (ETB-SimTog) method

Table 13 shows the result and computation of improvement of the *ETSB-SimTog* method over the MinMin. These results were yielded after running experiment 1 and experiment 5.

The MinMin used a total of 242033 and an average of 3486.2 ms to schedule the jobs, using two groups. ETB-SimTog used a total of 82557 and an average of 8255.7 ms to schedule the same task. Four groups used a total of 17569 and an average of 1756.9 ms to schedule the task, while eight groups used a total of 3587 and an average of 358.7 ms to schedule the same task. Figure 7 illustrates the total and average scheduling times. The ANOVA test was used to check the significance of the results, shown in Table 14. All differences were found to be highly significant, exhibiting very low P values.

Using two groups, the *ETSB-SimTog* method recorded between 2.36 and 4.07 speed-ups with an average of 3.21 speed-ups representing an average of 86.53 percent performance improvement over the MinMin. Using four groups, recorded between 10.78 to 17.07 speed-ups with an average of 14.58 speed-up representing 97.58 percent improvement over the MinMin. While eight groups recorded between 34.42 to 71.04 speed-up, averaging 63.97 and representing 99.28 percent performance improvement over the MinMin.

Across all the groups, there was a general improvement in the speed-up to a point after which the speed-up declines. Figures 8 and 9 show the scale of improvement of the *ETSB-SimTog* method in multiples and in percentage. Figure 9 shows improvement from 2 to 8 groups for the same method. Figure 10 shows the rate of improvement and we see that there is a levelling off of performance between the groups, demonstrating a possible limit to the amount of improvement that is achievable with successive groups. The calculation of the rates of improvement is shown in Table 15.

Table 13: Scheduling times and speed-up for MinMin vs. ETSB-SimTog

Methods	**MinMin Vs ETSB-SimTog Scheduling time in ms**				**Speed-up (X)**			**Speed-up (%)**		
Jobs Limit	**MinMin**	**2Grps**	**4Grps**	**8Grps**	**2Grps**	**4Grps**	**8Grps**	**2Grps**	**4Grps**	**8Grps**
1000	654	181	63	19	3.61	10.38	34.42	84.56	93.88	98.32
2000	3230	793	192	51	4.07	16.82	63.33	89.75	97.15	99.23
3000	7601	1876	447	110	4.05	17.00	69.10	89.92	97.86	99.39

4000	12920	3691	757	183	3.50	17.07	70.60	88.58	98.05	99.41
5000	18219	7706	1178	283	2.36	15.47	64.38	86.77	98.23	99.45
6000	22671	8576	1548	360	2.64	14.65	62.98	85.84	98.31	99.44
7000	29504	10343	2133	437	2.85	13.83	67.51	84.17	98.27	99.37
8000	39074	12399	2555	550	3.15	15.29	71.04	85.76	98.13	99.42
9000	48178	15984	3527	685	3.01	13.66	70.33	85.49	98.02	99.39
10000	59982	21008	5169	909	2.86	11.60	65.99	84.42	97.91	99.43
Total	**242033**	**82557**	**17569**	**3587**	**32.12**	**145.78**	**639.69**	**865.27**	**975.81**	**992.85**
Average	**24203.3**	**8255.7**	**1756.9**	**358.7**	**3.21**	**14.58**	**63.97**	**86.53**	**97.58**	**99.28**

Table 14: ANOVA results for ETSB-SimTog vs. MinMin and between groups

Test	Method	P Value	Significant Difference ?
1	MinMin/ ETSB-SimTog (All)	0.00423	Yes
2	MinMin/ ETSB-SimTog(2Grps)	0.0273	Yes
3	MinMin/ ETSB-SimTog(4Grps)	0.002202	Yes
4	MinMin/ ETSB-SimTog(8Grps)	0.001306	Yes
5	ETSB-SimTog(2Grps)/ ETSB-SimTog(4Grps)	0.00946	Yes
6	ETSB-SimTog(2Grps)/ ETSB-SimTog(8Grps)	0.001943	Yes
7	ETSB-SimTog(4Grps)/ ETSB-SimTog(8Grps)	0.015697	Yes

Table 15: Performance improvement for ETSB-SimTog method vs. MinMin and between groups

Methods		**ETSB-SimTog Performance Improvement(X)**			**ETSB –SimTog Performance Improvement (%)**			
	MinMin	**2Grps**	**4Grps**	**8Grps**	**Methods**	**2Grps**	**4Grps**	**8Grps**
Total	242033	82557	17569	3587		82557	17569	3587
$\frac{Total_{MinMin}}{Total_{Group}}$		2.93	13.78	47.48	$\frac{x_1 - x_2}{x_1} X100$ $x_{1 = \text{MinMin}}$	65.89 $x_{2 =2\text{Grps}}$	92.74 $x_{2 =4\text{Grps}}$	98.52 $x_{2 =8\text{Grps}}$
$\frac{Total_{nGrps}}{Total_{n+1Grps}}$ $n \in [2,4,8]$			4.70	23.02	$x_{1 = 2\text{Grps}}$		78.72 $x_{2 =4\text{Grps}}$	95.66 $x_{2 =8\text{Grps}}$

$\frac{Total_{nGrps}}{Total_{n+1Grps}}$ $n \in [2,4,8]$				4.90	$x_{1 = 4Grps}$			79.58 $x_{2 =8Grps}$

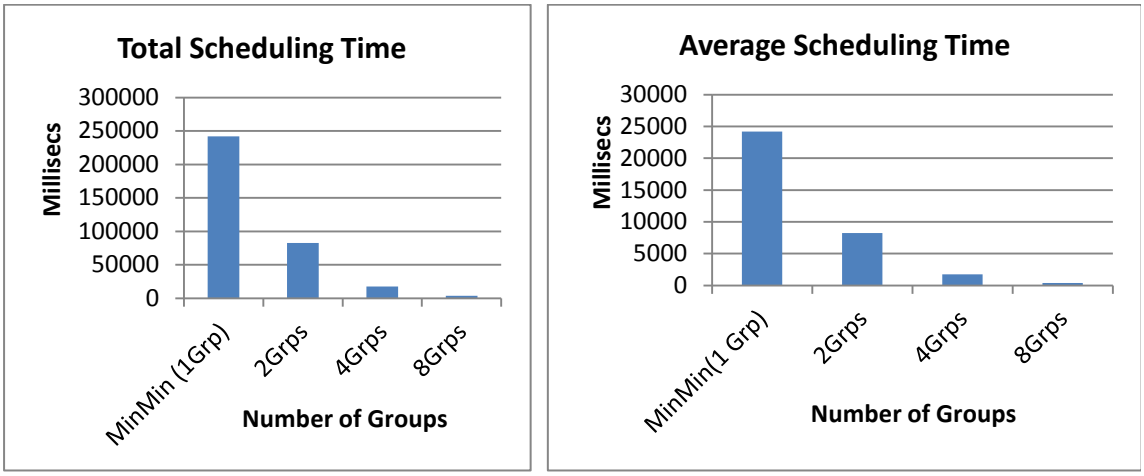

Figure 7: Total and average scheduling times of MinMin and across groups (ETSB-SimTog)

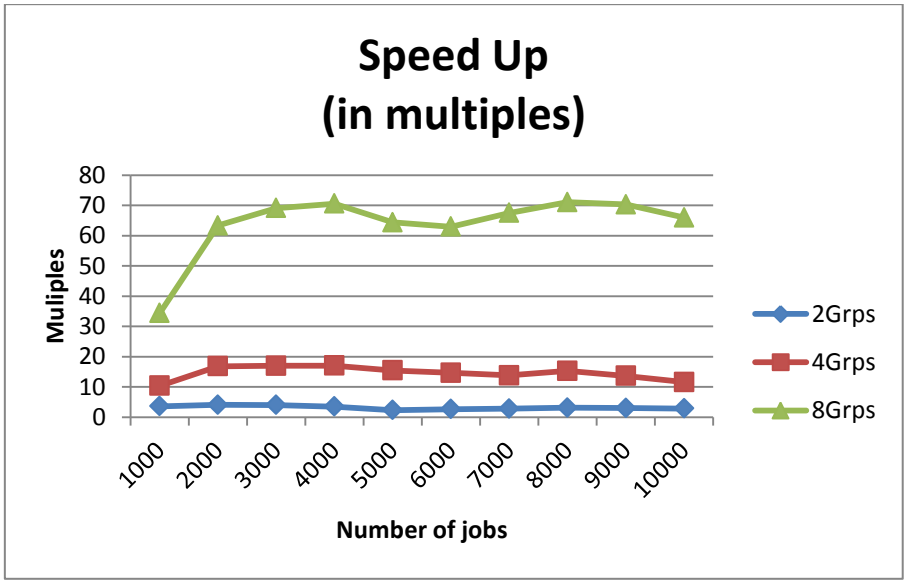

Figure 8: Speed-up (in multiples) of ETSB-SimTog over MinMin with increasing jobs

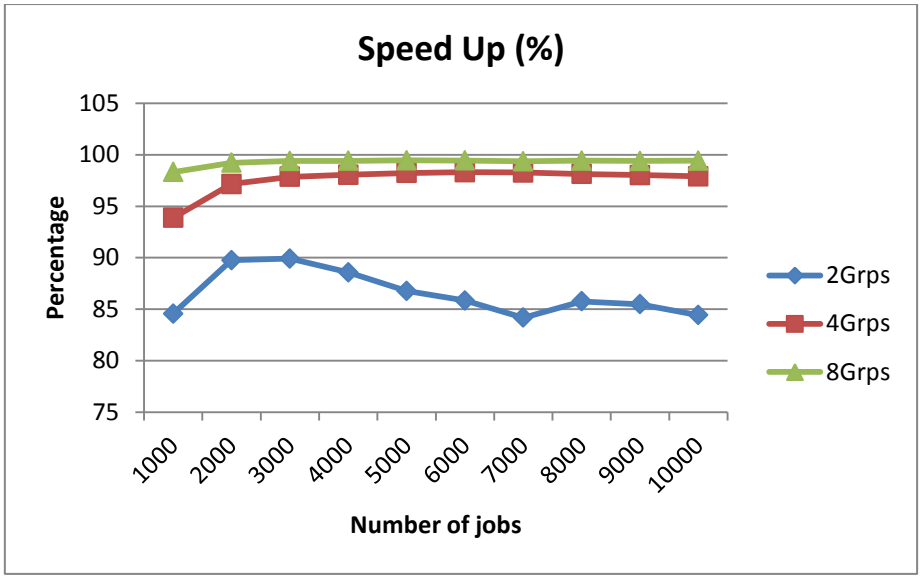

Figure 8: Speed-up (in percentage) of ETSB-SimTog over MinMin with increasing jobs

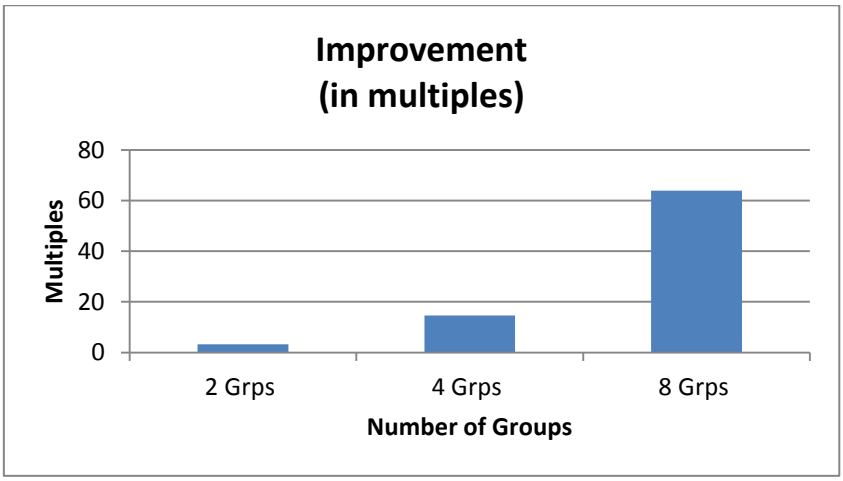

Figure 9: Improvement of ETSB-SimTog on MinMin across group cardinality

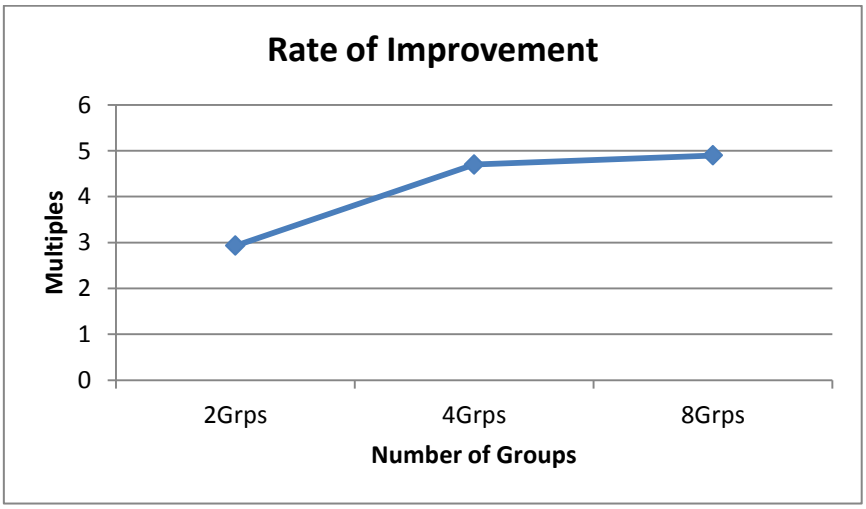

Figure 10: Speed-up rate as group cardinality increases (ETSB-SimTog)

6.2 Comparison of grouping methods

This section compares the performance of all the grouping methods against MinMin alone when using four groups and four threads. These results were yielded from experiments 1 to 5. We compared the average and total scheduling time of the *ETB* and *ETSB* methods of job grouping on both methods of machine grouping, *EvenDist* and *SimTog*. Table 16 shows the scheduling times and speed-up. The ANOVA significance test results for the various performances are shown in Table 17.

Figure 11 illustrates the difference in performance between MinMin and the grouping methods. We see that all the grouping methods perform much better than MinMin. The ANOVA results show these differences to be significant. Figure 12 compares the grouping methods without MinMin so that we can differentiate their performance more clearly. We see that the *ETSB-SimTog* method performs worse than the others. The ANOVA results, which are discussed in the next paragraph, show this performance difference to be significant. There was no significant difference between the performances of the other grouping methods.

All grouping methods performed better than MinMin with significant differences shown in the ANOVA results. Test 1 in Table 17 has taken the mean scheduling speed of all grouping methods (we call this GPMS) and compared this to MinMin and a significant difference is shown. This shows that overall our GPMS performs significantly better than MinMin alone. We also see than the *ETB* and *ETSB* methods perform significantly better than MinMin. The significance analysis shows that there was no difference between grouping methods *ETB* and *ETSB* (Table 17, Test 8), both of which perform significantly better than MinMin. However a difference is shown between the following methods: *ETB-EvenDist* vs. *ETSB-SimTog* (Test 10); *ETB-SimTog* vs. *ETSB-SimTog (*Test 12); and between *ETSB-EvenDist* vs. *ETSB-SimTog (*Test 14). We see that although *ETSB-SimTog* performs significantly better than MinMin (Table 17, Test 7), it performs significantly worse than the other grouping methods (Table 17, Tests 10, 12 and 14).

Table 18 shows the speed-up for all grouping methods. We see that for four groups, three of the methods (*ETB-EvenDist*, *ETB-SimTog*, *ETSB-EvenDist*) perform at speed-ups of over 46 compared to MinMin whereas *ETSB-SimTog* achieved a speed-up of 15 times over MinMin. There is a significant difference between this speed-up time and the others in the comparative set.

Figure 13 shows improvement across group cardinalities for all methods, showing increased speed-up as the number of jobs increases. Figure 14 shows the overall rate of improvement in multiples for all GPMS methods and here we see a decrease as the number of groups approaches 8, again showing a levelling-off of improvement rate.

Table 16: Scheduling times and speed-up for ETB vs. ETSB

Method		**ETB**		**ETSB**	
Jobs	**MinMin**	**EvenDist**	**SimTog**	**EvenDist**	**SimTog**
1000	654	40	32	24	63
2000	3230	92	50	61	192
3000	7601	163	99	119	447
4000	12920	252	196	186	757
5000	18219	323	324	333	1178
6000	22671	383	522	518	1548
7000	29504	511	703	532	2133
8000	39074	729	907	744	2555
9000	48178	954	992	949	3527
10000	59982	1254	1399	1177	5169
Total	242033	4701	5224	4643	17569
Ave	24203.3	470.1	522.4	464.3	1756.9
Performance Speed-up (X)		51.48	46.33	52.13	13.78

Table 17: ANOVA results for MinMin and all grouping methods

Test No	Method	P value	Significant Difference? (Threshold level: P = 0.05)
1	MinMin vs. GPMS	0.001537	Yes
2	MinMin vs. ETB	0.001373	Yes
3	MinMin vs. ETSB	0.001723	Yes
4	MinMin vs. ETB-EvenDist	0.00136	Yes
5	MinMin vs. ETB-SimTog	0.001387	Yes
6	MinMin vs. ETSB-EvenDist	0.001357	Yes
7	MinMin vs. ETSB-SimTog	0.010622	Yes
8	ETB vs. ETSB	0.093828	No
9	ETB-EvenDist vs. ETSB-EvenDist	0.974201	No
10	ETB-EvenDist vs. ETSB-SimTog	0.026158	Yes
11	ETB-SimTog vs. ETSB-EvenDist	0.767002	No
12	ETB-SimTog vs. ETSB-SimTog	0.033619	Yes
13	ETB-EvenDist vs. ETB-SimTog	0.790165	No
14	ETSB-EvenDist vs. ETSB-SimTog	0.025532	Yes
15	EvenDist vs. SimTog	0.073511	No

Table 18: Grouping method, group cardinality and speed-up

	Speed-up		
Grouping Method	**2 Groups**	**4 Groups**	**8 Groups**
ETB -EvenDist	7.62	47.46	155.33
ETB-SimTog	6.98	46.33	157.06
ETSB-EvenDist	6.79	52.13	190.57
ETSB-SimTog	3.21	14.58	63.97

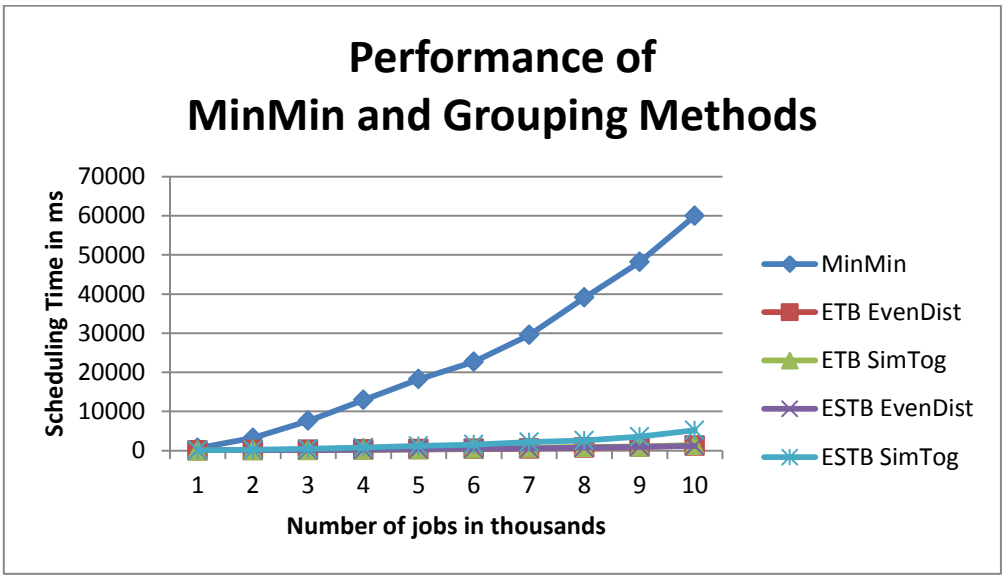

Figure 11: Performance comparison of MinMin and grouping methods

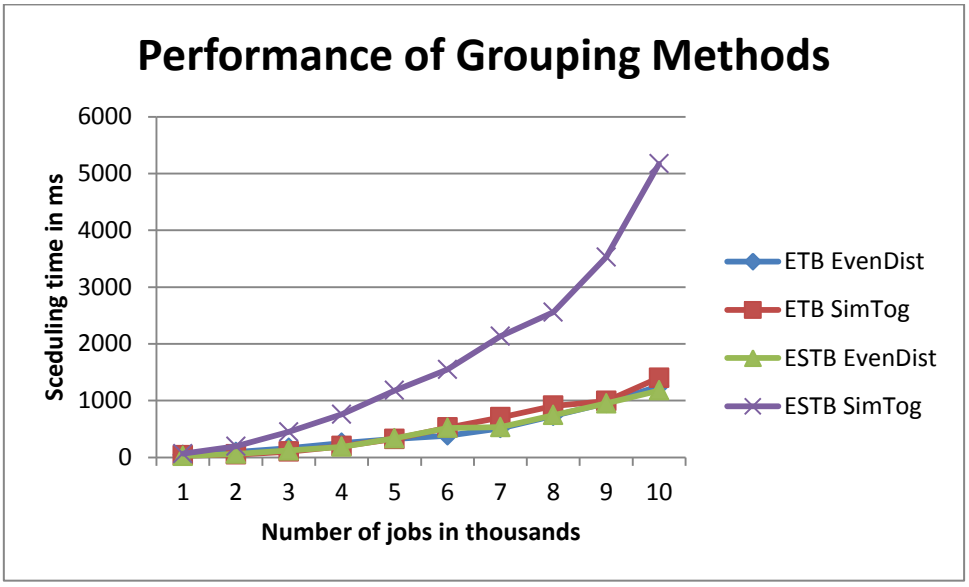

Figure 12: Performance comparison of grouping methods

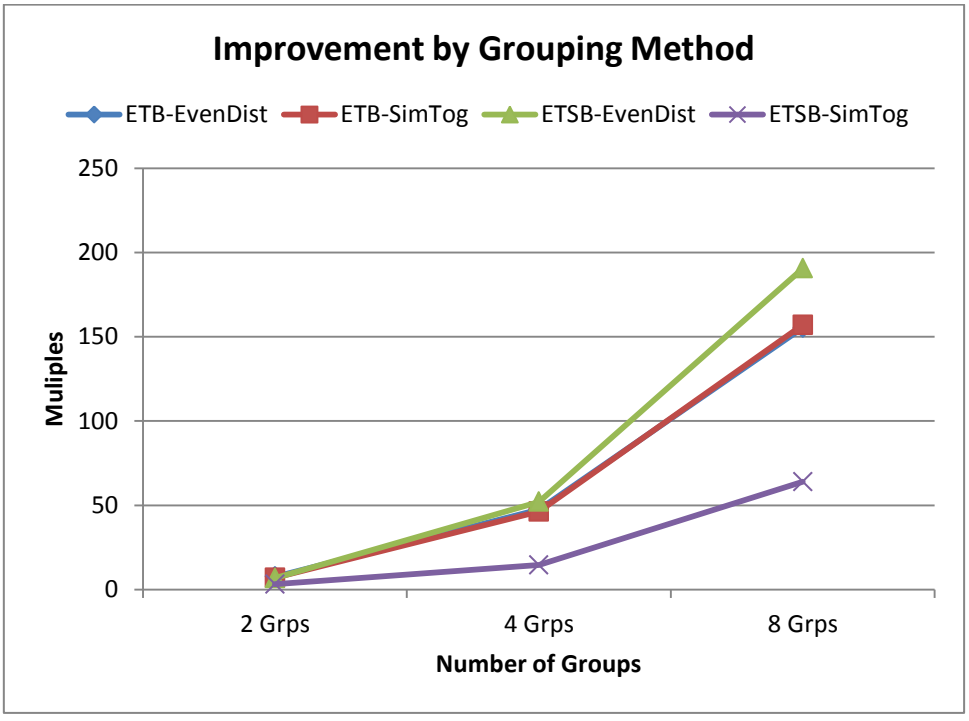

Figure 13: Improvement across methods

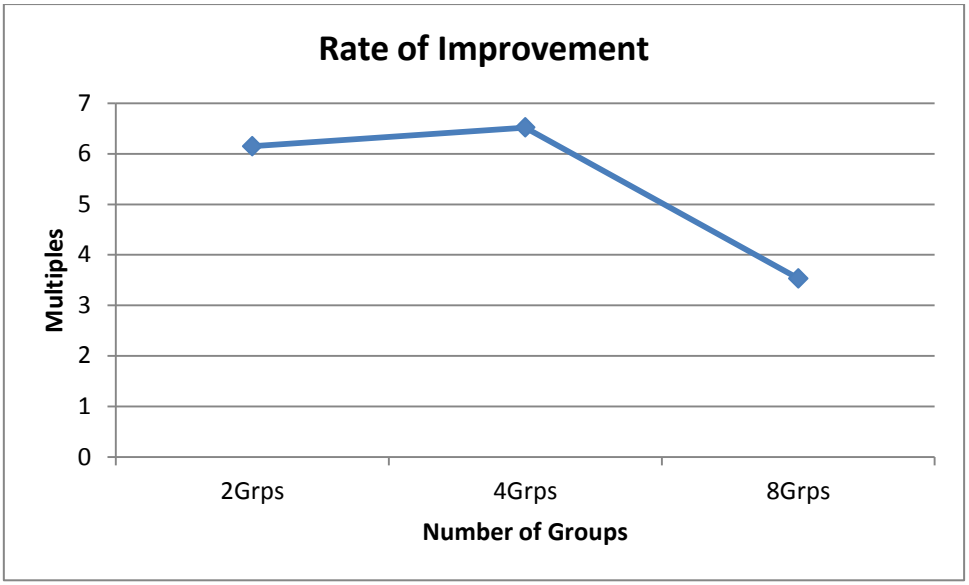

Figure 14: Overall rate of improvement with increasing group cardinality

6.3 Comparison between GPMS methods and the Priority method

This section compares the results of the two GPMS grouping methods (*ETB* and *ETSB*) against results of the *Priority* method (PPMS) from our previous work [1].

Results for this comparison are from 4 groups. Our *Priority* method used 4 groups so the selection of the group cardinality of 4 allows fair comparison to be made. The number of threads was also 4 in all tests. Table 19 provides the raw results. Figure 15 illustrates the difference between the various methods and Figure 16 shows the overall difference between *Priority*, *ETB* and *ETSB*. Table 20 shows the significance through ANOVA testing.

With the *Priority* job grouping method, the *SimTog* and *EvenDist* methods recorded a performance improvement of 5.90 and 6.76, with times of 41006 and 35807 ms respectively over the MinMin. With the *ETSB* grouping method, the *SimTog* and *EvenDist* methods recorded a 13 times and 52 times performance improvement over the MinMin using 17569 and 4643 ms respectively, while the *ETB* job grouping method yielded 46 and 51 times performance improvement between the *SimTog* and *EvenDist* methods respectively using 5224 and 4701 ms respectively to perform the scheduling task. The ANOVA results which show the significance of the performance differences are given in Table 20. We have compared *Priority* to *ETB* and *ETSB* for both machine grouping methods (*EvenDist* and *SimTog*). We also have compared *Priority* with *ETB* and *ETSB* separately regardless of machine grouping method. Furthermore, we have also compared *Priority* with *ETB* and *ETSB* together as a combined group. We call this group the GPMS group and its scheduling time is the mean of the scheduling times for *ETB* and *ETSB*.

We see in Figures 15 and 16 that the *ETB* and *ETSB* method generally perform much better than the *Priority*. The ANOVA test results generally back up this observation (see Table 20). The only exception is *Priority* vs. *ETSB*, where the significance is marginal, right on the $P<0.05$ boundary (see Table 20, Test 3). If we look more closely at this we see that the problem is the method *ETSB-SimTog* where the improvement is much smaller than the other GPMS methods. In fact there is no significant difference between *Priority-SimTog* and *ETSB-SimTog* (see Table 20, Test 7). In Figure 15, we can see that the performance of the *ETSB-SimTog* method is much closer to that of the *Priority* methods than any other GPMS method. Overall though, we can conclude that the GPMS methods perform better than the

Priority method. The ANOVA analysis of *Priority* method versus GPMS methods combined gave a P value of 0.027992 which shows that the difference is significant (see Table 20, Test 1). Figure 17 illustrates the difference between *Priority* and GPMS methods in the form of a chart.

The reason the GPMS methods performed much better than *Priority* is because in our previous experimentation with the *Priority* method we found that priorities were often not evenly distributed, resulting in some priority groups being much larger than others. , Since the MinMin scheduling algorithm tends to polynomial [23], larger groups have a comparatively inflated scheduling time requirement. Although in some cases *Priority* might work equally well as *ETB* or *ETSB*, this cannot be guaranteed unless the priority allocations are known in advance or balanced by a priority-allocation algorithm.

Table 19: Scheduling times and speed-up for Priority, ETB and ETSB

Method		**Priority**		**ETB**		**ETSB**	
Jobs	**MinMin**	**EvenDist**	**SimTog**	**EvenDist**	**SimTog**	**EvenDist**	**SimTog**
1000	654	95	105	40	32	24	63
2000	3230	340	412	92	50	61	192
3000	7601	673	839	163	99	119	447
4000	12920	1092	1345	252	196	186	757
5000	18219	1776	2008	323	324	333	1178
6000	22671	2837	3339	383	522	518	1548
7000	29504	3860	4570	511	703	532	2133
8000	39074	5312	7500	729	907	744	2555
9000	48178	7818	8830	954	992	949	3527
10000	59982	12004	12058	1254	1399	1177	5169
Total	242033	35807	41006	4701	5224	4643	17569
Ave	24203.3	3580.7	4100.6	470.1	522.4	464.3	1756.9
Performance Improvement(X)		6.76	5.90238	51.48	46.33	52.13	13.78

Table 20: ANOVA results for Priority and all grouping methods

Test	**Method**	**P value**	**Significant Difference ?**
1	Priority vs. GPMS (ETB and ETSB averaged)	0.027992	Yes
2	Priority vs. ETB	0.015965	Yes
3	Priority vs. ETSB	0.048583	Marginal
4	Priority-EvenDist vs. ETB-EvenDist	0.020335	Yes
5	Priority-SimTog vs. ETB-SimTog	0.013124	Yes
6	Priority EvenDist vs. ETSB-EvenDist	0.020128	Yes
7	Priority SimTog vs. ETSB-SimTog	0.109315	No

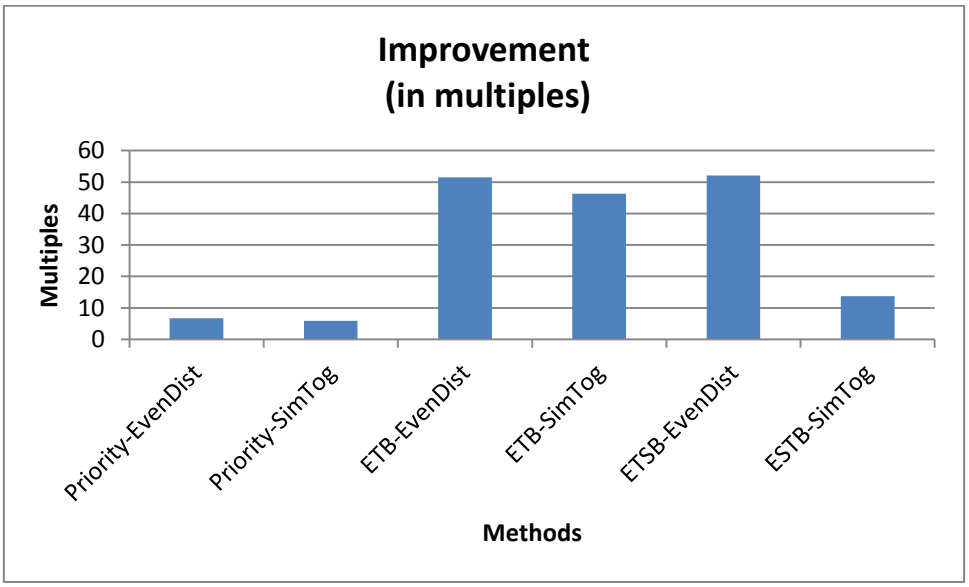

Figure 15: Improvement comparison between GPMS methods and Priority methods

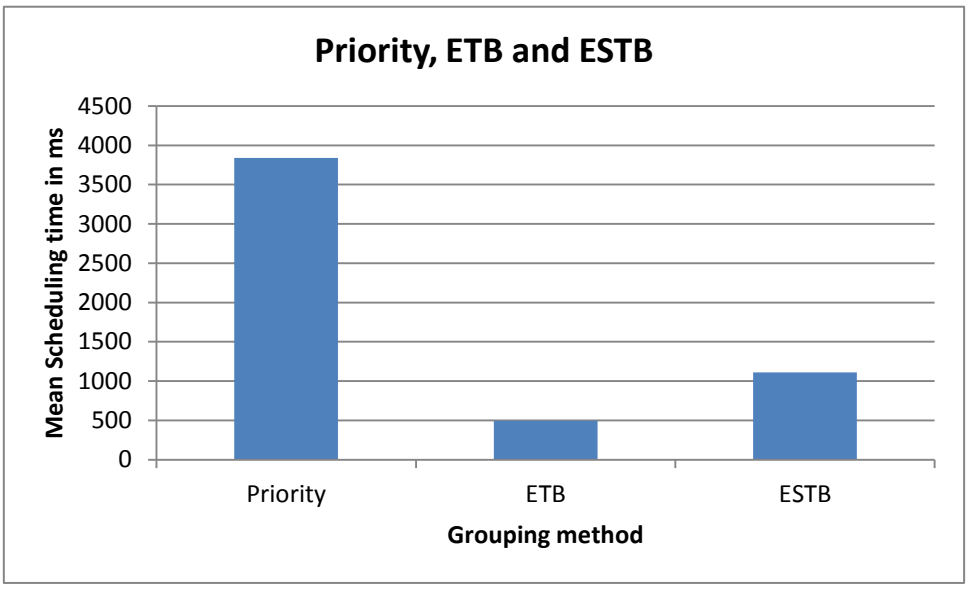

Figure 16: Priority method versus ETB and ETSB methods

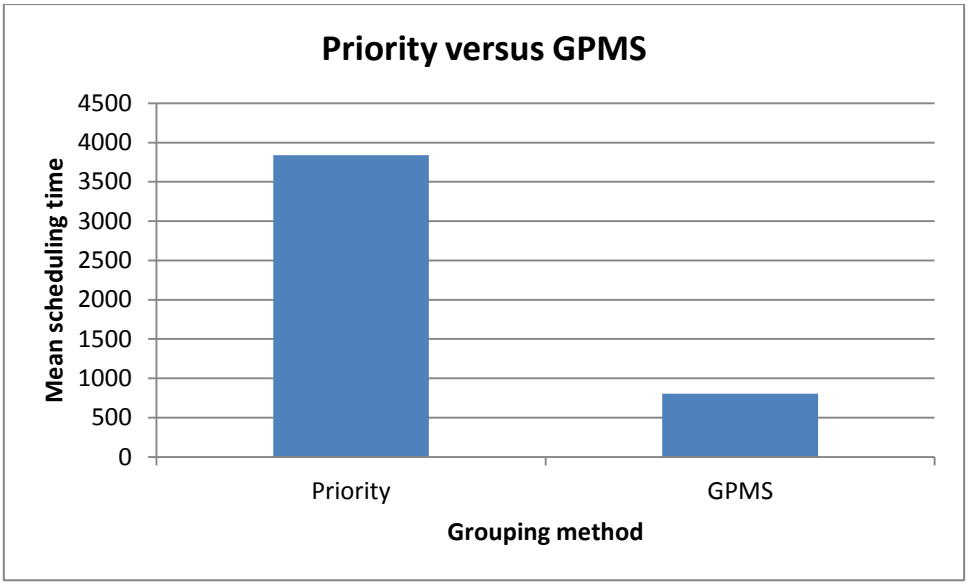

Figure 17: Priority versus GPMS

7. Discussion

We have explored job grouping methods in a bid to increase throughput in scheduling Grid jobs by exploiting the multicore hardware. Two methods (*ETB* and *ETSB*) were used to group jobs before scheduling in parallel. Two machine grouping methods (*Similar Together* and *Evenly Distributed)* were also used to group jobs. Parallelism in scheduling was executed using dynamic threads. In the first experiments, the MinMin scheduling algorithm was implemented alone, then in separate but subsequent experiments, the grouping methods were first used to group the jobs and machines, before implementing the MinMin algorithm again to schedule same range of jobs. The dataset for the experiment was taken from the Grid Workload Archive (GWA) [22].

In each of the experiments, we executed the MinMin algorithm on an HPC to schedule a range of jobs (from 1000 jobs to 10000 jobs in steps of 1000). The range of jobs were kept between 1000 and 10000. We experimented in steps of 1000 so we that we could determine the effect of increasing jobs on the speed-up by the method, and also to ease the various computations because computations are easier done with 10s, 100s, 1000s etc. In this paper we have reported on our experimentation with 1, 2, 4 and 8 groups using 4 threads. In the complete experimentation we varied both groups and threads between 1 to 16 in steps of power 2 (2^n). This is because multicore computers exist in that order and also so we can try to establish the relationship between the number of groups used, number of threads used and number of CPUs used. However because of the number of combinations of variables yielding extensive results we are unable to report on all findings at this stage. Our future work will investigate further into these aspects and will report on effects of group, CPU and thread variation.

Our results showed that increasing the number of groups with our method increases the efficiency of Grid scheduling by large margins. By grouping jobs and executing the scheduling in parallel within the groups, we found that scheduling of Grid jobs improved by 2 to 7 times when using two groups to schedule. With four groups, scheduling improved by 13 to 51 times and when using eight groups, scheduling improved by 59 to 253 times. Percentage-wise, our results showed that using two groups improved the scheduling efficiency by 81 to 87 percent. Four groups improved the efficiency of scheduling by 97 to 98 percent while eight groups increased the performance by up to 99 percent. Between the groups, there was 80 to 84 percent improvement using four groups over two groups and

between 67 and 69 percent improvement by eight groups over four groups. There was a pattern exhibited by the performance graph in all the cases. The pattern was that the rate of improvement increases up to a point and then levels off. This is due to the relationships between threads, cores and groups in the run-time environment and associated overheads.
With low numbers of groups, the overheads have a relatively large effect but their effect diminishes as the number of groups increases and higher improvements are gained until a steadier state and rate of improvement emerges. However the number of threads and the way the parallelisation is carried out across the cores can disturb the steady state. Further investigation is required to establish a robust theoretical model of the relationships in the dynamic run-time environment.

We showed that grouping of jobs can be exploited in improving Grid scheduling by comparing our two methods of job grouping (*ETB* and *ETSB*) against each other, the *ETB* method performed similarly to the *ETSB* method when using *EvenDist* because both machines and jobs were fairly distributed in this case. The *ETB* performed better than the *ETSB* method when using *SimTog* to group machines. In fact of the four grouping combinations, *ETB-EvenDist*, *ETB-SimTog*, *ETSB-EvenDist*, and *ETSB-SimTog*, the former three all performed significantly better than the latter one according to our results. At the moment we do not have a definite explanation for this observation.

We compared the GPMS methods to our *Priority* method [1]. The *Priority* method performed less well than the other GPMS methods because in our data set the jobs were not equally spread among the priority groups. It happened that a large number of jobs were assigned to the same machine group. The polynomial-time MinMin algorithm for scheduling within groups therefore took relatively longer, increasing the overall scheduling time disproportionally. If the priority job groups had been evenly balanced, the poorer performance would not have occurred. The makespan also increases if the jobs groups are not balanced because the heavier loaded group would take longer to complete.

It will be noted that neither PPMS nor GPMS targets the specialised GPU environment as most other research in this area does. Both methods execute correctly on general purpose systems – including HPC, and standalone computers. This is intended to widen the scope of applicability of the method in scheduling of not just Grid jobs. The approach can be extended to GPUs, distributed systems and to any Grid or Cloud environment which is characterized by a requirement to schedule a stream of jobs.

After jobs from the GWA trace file are read into the scheduler, two different methods were used to split the jobs equally into the specified number of groups. Both methods depended on the estimated times for the jobs. Estimates of execution times are computed from the size of jobs which are also computed from the attributes. After the estimation has been made for all jobs, the information is held in a table to be used for scheduling later. We recognise that the estimated processing times are affected depending on the trace file attributes used by the estimation method. In fact, it is not possible to generate the accurate job sizes from the GWA trace file because some values are missing, including machine speed of the original machines. However this limitation does not affect the validity of our results since the usefulness of the trace file for us was as a source from which a variety of job data could be generated. The relationship of our generated data to that of the original jobs was not crucial. The two methods used in distributing the jobs in the GPMS approach balance the jobs equally into the groups hence the distribution of the job attributes from the trace file would theoretically have no direct impact on scheduling times. However, it would have an impact on the execution times of jobs and hence on the makespan.

8. Conclusion and future work

The contribution of this work has been to show how different types of grouping can harness parallelism in multicores and positively affect scheduling speed. The resulting Group Parallel Multi-scheduler (GPMS) can be used in any environment in which there is a requirement to schedule a stream of jobs onto a set of limited resources. Typical environments which could benefit are Grid and Cloud environments. Given the growth in these paradigms, the research has potential to be exploited widely.

The interest of most researchers in Grid scheduling has been on creating schedules such that overall makespan is decreased. This research improves on those efforts by providing a method by which the scheduling can also be carried out in parallel and thus overall makespan can be decreased further. We have proposed the GPMS which can be configured with varying grouping methods to suit varying characteristics of incoming jobs. Our experimentation has shown that idiosyncrasies of the input job set can have an effect on the time taken to schedule, depending on the scheduling method used. It can also affect the quality of the resulting schedule. Thus, the best results might be obtained by using an adaptive GPMS which can exploit different scheduling mechanisms depending on the characteristics of the incoming jobs. Future work will further explore the relationship between the job characteristics, machine characteristics, scheduling method, grouping parameters, scheduling time and makespan. It will investigate alternative grouping methods and how characteristics of input jobs can be identified early and exploited such that appropriate grouping methods can be selected based on job and machine characteristics in an adaptive GPMS.

In our experiment the threads were not explicitly bound to CPUs. Thus we did not exercise explicit control over the 12 cores on the compute node of the HPC. Such control would have offered us an opportunity to investigate the relationship between increasing number of groups and increasing number of CPUs relative to scheduling efficiency. Future work should include a means of varying the CPUs on the HPC machine just the same way the number of groups were varied and investigating the relationship between both and also in the context of varying numbers of threads.

In very complex environment, it will be interesting to extend this study of parallel multi-scheduling on multicores by the implementation of several (or different) scheduling algorithms across the different job-machine groups. That is, we can decide to independently execute a mix of different scheduling algorithms from each of the independent groups. This will enable us to use one scheduling algorithms that suits jobs in one group and use another scheduling algorithm which suits another set of jobs in another group. If characteristics or attributes of certain jobs affect the efficiency of the scheduling method, then this will provide the opportunity to exploit the benefits of one scheduling algorithm (from one set of jobs in one group) against the disadvantages of the other (in another set of jobs in another group). That means, we can implement a scheduling algorithm from the groups based on what scheduling method favours jobs in that group.

Finally, the experiment was executed in a simulated environment and not on a real test bed, while the differences of a simulated environment and that of a real system or test bed are out of the scope of this work, it will be worthwhile to state here that effort should be made in due course to test the scheduler on a real test bed to ascertain the real functionality of the method.

Acknowledgement

We are grateful to the following groups and members of their teams for making the Grid Workload Archive available and free to researchers and developers alike:

- The Parallel and Distributed Systems Group at Delft University of Technology (TUDelft), Netherlands (GWA 2014) (http://www.pds.ewi.tudelft.nl/). Members of the group are: Shanny Anoep (TU Delft); Catalin Dumitrescu (TU Delft); Dick Epema (TU Delft); Alexandru Iosup (TU Delft); Mathieu Jan (TU Delft); Hui Li (U. Leiden); and Lex Wolters (U. Leiden).
- The e-Science Group of HEP at Imperial College London for providing the LCG data and Hui Li for making the data publicly available and Dr Feitelson of the parallel workloads archive. http://lcg.web.cern.ch/LCG
- The Grid'5000 team (especially Dr. Franck Cappello) and the OAR team (especially Dr. Olivier Richard and Nicolas Capit) for the trace http://oar.imag.fr . Also special thanks to John Morton (john_x_sharrcnet.ca) for providing the trace file and for making the parallel workload archive publicly available
- The AuverGrid team with special thanks to Dr. Emmanuel Medernach, the owner of the AuverGrid system made available through the Grid workloads archive http://auvergrid.fr
- NorduGrid team, with special thanks to Dr. Balasz Knoya, the owner of NorduGrid system made public through the Grid workload archive.

References

[1] G. T. Abraham, A. James, N. Yaacob, Priority-grouping method for parallel multi-scheduling in grid, Journal of Computer and System Sciences, 2015 (in press).

[2] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, 1979.

[3] M. Maheswaran, M., S. Ali, H. J. Siegel., D.Hensgen., and R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 59 (1999) 107-131.

[4] W. Mu'alem, D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, Transactions on Parallel and Distributed Computing, 12 (2001) 529-543.

[5] B. G. Lawson, E. Smirni, Multiple-queue backfilling scheduling with priorities and reservations for parallel systems, in: Job Scheduling Strategies for Parallel Processing, Springer, Berlin Heidelberg, 2002, pp. 72-87.

[6] G. Sabin, R. Kettimuthu, A. Rajan, P. Sadayappan, Scheduling of parallel jobs in a heterogenuous multi-site environment, in: Job Scheduling Strategies for Parallel Processing. Springer, Berlin Heidelberg, 2003, pp. 87-104.

[7] W. Zhang, Albert M. K. Cheng, Multisite co-allocation algorithms for computational grid, in: Proceedings of the 20th International Parallel and Distributed Symposium Symposium (IPDPS), IEEE, 2006, pp. 8-pp.

[8] A. Quezada-Pina, A. Tchernykh, J. L. Gonzalez-Garcia, A. Hirales-Carbajal, J. M. Ramirez-Alcaraz,U. Schwiegelshohn, R. Yahyapour, V. Miranda-López, Adaptive parallel job scheduling with resource admissible allocation on two-level hierarchical grids, Future Generation Computer Systems, 28 (2012) 965-976.

[9] F. Liang, Y. Liu, H. Liu, S. Ma, B. Schnor. A parallel job execution time estimation approach based on user submission patterns within computational grids, International Journal of Parallel Programming (2013) 1-5.

[10] M. Kalantari, K. M. Akbari, A parallel solution for scheduling of real time applications on grid environments, Future Generation Computer Systems, 25 (2009) 704-716.

[11] J. Wang, B. Bin, H. Liu, L S. Li, J. Yi, Heterogeneous computing and grid scheduling with parallel biologically inspired hybrid heuristics, Transactions of the Institute of Measurement and Control (2014)

[12] J. Chen, B. Li, E. F. Wang, Parallel scheduling algorithms investigation of support strict resource reservation from grid, Applied Mechanics and Materials 519 (2014) 108-113.

[13] R. Albodour, A. James, N. Yaacob, High level QoS-driven model for grid applications in a simulated environment, Future Generation Computer Systems 28 (2012) 1133-1144.

[14] P. Xiao, L. Dongbo, Multi-scheme co-scheduling framework for high-performance real-time applications in heterogeneous grids, International Journal of Computational Science and Engineering 9 (2014) 55-63.

[15] S. Nesmachnow, M. Canabé, GPU implementations of scheduling heuristics for heterogeneous computing environments, in: XVII Congreso Argentino de Ciencias de la Computación, 2014.

[16] M. Canabé, S. Nesmachnow,Parallel implementations of the MinMin heterogeneous computing scheduler in GPU, CLEI Electronic Journal 15 (2012) 8-8

[17] F. Pinel, B. Dorronsoro, P. Bouvry, (2013), Solving very large instances of the scheduling of independent tasks problem on the GPU, Journal of Parallel and Distributed Computing, 73 (2013) 101-110.

[18] S. Nesmachnow, H. Cancela, E. Alba, Heterogeneous computing scheduling with evolutionary algorithms, Soft Computing, 5 (2010) 685-701.

[19] S. Nesmachnow, E. Alba, H.Cancela, Scheduling in heterogeneous computing and grid environments using a parallel CHC evolutionary algorithm, Computational Intelligence 28 (2012) 131-155.

[20] S.Nesmachnow, H. Cancela, E. Alba, A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling, Applied Soft Computing. 12 (2012) 626-639.

[21] S. A. Mirsoleimani, A. Karami, F. Khunjush, A parallel memetic algorithm on GPU to solve the task scheduling problem in heterogeneous environments, in: Proceedings of the Fifteenth Annual Conference on Genetic and Evolutionary Computation, ACM, 2013, pp. 1181-1188.

[22] GWA 2014 , The Grid Workload Archive available from <http://gwa.ewi.tudelft.nl/> [accessed 3rd October 2014]

[23] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, t. Kidd, M. Kussaw, J.D. Luna, F. Mirabile, L. Moore, B. Rust, H.J. Siegel, Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet, in : Proceedings of the Seventh Heterogeneous Computing Workshop, 1998 (HCW 98) IEEE, 1998, pp. 184-199.