The Greedy Multi-cluster Scheduler: Performance Bounds and Parametric Sensitivity

John Ngubiri and Mario van Vliet

Nijmegen Institute for Informatics and Information Science
Radboud University Nijmegen
Toernooiveld 1, 6525 ED, Nijmegen,
The Netherlands
{ngubiri, mario}@cs.ru.nl

Abstract. Most schedulers in parallel job scheduling do not put (job) schedulability into consideration when prioritizing jobs. Performance evaluation is mostly done using average values of the measurement metric. Using the average metric value may conceal relative job starvation details hence giving a shallower understanding of scheduler performance. We propose a greedy multi-cluster scheduler that uses the (estimate of) job schedulability and the time a job has spent in the queue to compute its priority. We compare the performance of our scheduler with that of Fit Processor First Served (FPFS) scheduler. We also study the sensitivity of its performance to parameter changes. We observe that (i) within some parameter ranges, our scheduler outperforms FPFS; (ii) for big jobs, our scheduler outperforms FPFS; for small jobs, FPFS outperforms our scheduler and (iii) our scheduler is fairer than FPFS.

1 Introduction

Clusters are very popular supercomputing platforms. Over 70% of the world’s top 500 supercomputers today are clusters [1]. A lot of work has been done on supercomputers [9], grids [11] as well as multi-cluster systems [4][13]. A multi-cluster system is set up by connecting multiple clusters (possibly in different geographical locations) into a bigger computational infrastructure. Unlike the grid, a multi-cluster system is less heterogeneous and comparatively small.

Different clusters, owned by different organizations, can be joined into a multi-cluster system. This provides each of the organizations with high computational power at a low extra cost. Cluster owners submit jobs to the system for processing. A job may be processed by any of the participating clusters. Large jobs may be broken into components and each component is processed in a separate cluster (co-allocation) [6]. Components of a co-allocated job start and finish processing at the same time. For a multi-component job, components communicate across the wide area network (WAN) that connects the clusters. This increases the execution time of the job beyond what it would take if processed in a single cluster. Bucur and Epema [5] show that despite the communication bottleneck, co-allocation still gives a performance benefit.
In this paper, we propose a multi-cluster scheduler that uses the estimate of job schedulability and the time spent in the queue to prioritize jobs. We compare its performance with FPFS and study its performance sensitivity to its parameters. We use the entire job stream and size based groups to evaluate performance. Group-wise evaluation [20][19][10][17] gives a better understanding of scheduler performance.

Within some parameter ranges, our scheduler outperforms FPFS. For both schedulers, small jobs perform better than large jobs. However, FPFS outperforms our scheduler for small jobs and our scheduler outperforms FPFS for large jobs. The performance difference between small and large jobs is higher in FPFS than in our scheduler. Therefore, our scheduler is fairer than FPFS.

The paper is organized as follows: We discuss the related work in Section 2, describe the research model in Section 3, and describe the scheduling algorithms together with the placement policy used in Section 4. In Section 5, we describe the experimental set up, the statistical distributions and parameters used. In Section 6, we compare the performance of selected scheduler instances and study performance sensitivity to parameters in Section 7. We then make conclusions and suggestions for future research in Section 8.

2 Related Work

Many schedulers have been proposed for parallel job scheduling [9]. For space slicing cases, backfilling is the most popular. Generally, backfilling allows some jobs to be processed outside the FCFS order. Conservative backfilling [15] allows a job to be processed outside the FCFS order if the reservation times of other jobs in the queue are not delayed. Aggressive backfilling [16] allow jobs to be processed outside the FCFS order if the reservation time of the job at the head of the queue is unaltered. Shmueli and Feitelson [18] improve backfilling by allowing the scheduler to look ahead so that the job that best utilizes the available processors is backfilled. Backfilling is possible if both job size and duration are known. Some job characteristics, however, are unknown in some scheduling environments (like in [7]). Lee et al. [14] showed that users are unable to accurately estimate their jobs’ runtimes. Most earlier work on scheduling in multi-cluster systems [3][4][13] did not assume prior knowledge of job runtime.

The FPFS algorithm allows jobs to jump and get processed without considering the effect on other jobs’ reservations. Starvation is controlled by limiting the number of times a job at the head of the queue is jumped. Studies by Ngubiri and van Vliet [17] show that though the indiscriminate jumping does not create a net disadvantage to large jobs, there is a big performance difference between large and small jobs.

Performance metrics used have to be chosen with a lot of care [12]. Average waiting time (AWT) and average response time (ART) for example give a similar performance implication for dedicated non pre-emptive scheduling cases but give different implications in preemptive or time slicing cases.
Since different jobs have different levels of schedulability [17], evaluating performance by job groups helps in getting a deeper understanding of scheduler performance. Srinivasan et al. [20] use job groups to get a deeper understanding of scheduler robustness for moldable jobs. Deeper comparative studies between conservative and aggressive backfilling were also studied in [19] by group-wise evaluation. Feitelson [10] used job groups to study performance implications for different metrics and workloads.

3 Research Model

We consider a system made up of $N$ clusters $C_1, C_2, \ldots, C_N$. Cluster $C_i$ has $n_i$ identical processors. The clusters process jobs in a dedicated manner. The clusters are interconnected with WAN links which have the same bandwidth. The system is served by one queue and one scheduler.

Jobs are rigid and online. The number of processors a job needs (size) is unknown until it arrives and the execution time is unknown until it finishes execution. If the job size is large (beyond a threshold value $\text{thres}$), it is broken into components and co-allocated.

The job sizes are generated from a distribution $D(q)$ defined over an interval $[n_1, n_2]$ ($0 < n_1 < n_2$). In $D(q)$, the probability $p_i$ that a job has size $i$ is given by $\frac{q^i}{Q}$ if $i$ is a power of 2 and given by $\frac{q^i}{Q}$ if $i$ is a power of 2. The parameter $q$ ($q < 1$) is used to vary the mean job size and $Q$ is in such a way that $p_i$ sums up to 1. The distribution favors small jobs and those whose size is a power of 2 which is known to be a realistic choice [8].

If a job has a big size ($\text{size} > \text{thres}$), it is broken into components. To break a job of size $s$ into $n$ components, the system makes the first $n-1$ components to have a width of $\lfloor \frac{s}{n} \rfloor$ each and the $n^{th}$ component to have a width of $s-(n-1)\lfloor \frac{s}{n} \rfloor$.

4 Scheduling Algorithms and Placement Policy

We use FPFS and our proposed greedy scheduling algorithm.

In FPFS, jobs are queued in their arrival order. The scheduler starts from the head and searches deeper into the queue for the first job that fits into the system. In case one is found, it jumps all jobs ahead of it in the queue and starts processing. If none is found, the scheduler waits either for a job to finish execution or a job to arrive and the search is done again. To avoid possible starvation of some jobs, the scheduler limits (to $\text{maxJumps}$) the number of times a job at the head of the queue is jumped. After being jumped $\text{maxJumps}$ times, no other job is allowed to jump it until enough processors has been freed (by terminating jobs) to have it scheduled. We use FPFS($x$) to represent FPFS when $\text{maxJumps} = x$.

Our greedy scheduler aims at giving unschedulable jobs earlier scheduling opportunities. Previous studies [20][19][10] show that jobs perform differently
depending on their characteristics. Our studies in [17] show that job schedulability highly depend on size \( w \) and width of the widest component \( w^* \). We therefore assume a hardness function \( h(w, w^*) \) that represents schedulability.

To prioritize jobs, we use the priority indicator which is got by multiplying the value of the hardness function for a job and the time the job has spent in the queue. Scheduling attempts are made in reducing order of the priority indicator. In case the job with the highest indicator cannot fit, the next is tried. To enforce fairness, we search for candidate jobs with in a certain depth in the queue and limit (to \( \text{maxJumps} \)) the times a job at the head of the queue can be jumped.

In this work, we assume that the hardness function is a linear function \( h(w, w^*) = \alpha w + \beta w^* \) (\( \alpha \) and \( \beta \) are constants). We assume \( \alpha = \beta = 1 \); varying \( \alpha \) and \( \beta \) do not produce substantial changes in performance.

The steps in the greedy scheduler are described below:

1. Check the times a job at the head of the queue has been jumped
   1.1 If it is jumped less than \( \text{maxJumps} \), go to step 2
   1.2 If it is jumped \( \text{maxJumps} \) times, check if it can fit in the system.
   1.3 If it can fit in the system, start its execution and go to 1
   1.4 If it cannot fit, wait until enough space is created in the system start its execution and go to 1.

2. Compute the priority indicators for the first \( \text{depth} \) jobs
3. In reducing order of indicators, check for the first job fits in the system.
   3.1 If a job is found, schedule it
   3.2 If none fits, wait until a job finishes execution and repeat step 2
4. If the job scheduled was not from the head of the queue, increment the number of times the job at the head of the queue is jumped by 1
5. Repeat the process starting from 1 until all jobs are finished.

We use \( \text{Greedy}(j, d) \) to represent the greedy scheduler when \( \text{maxJumps} = j \) and \( \text{depth} = d \).

To allocate components to clusters, we use the Worst Fit (WFit) policy. In the WFit policy, the \( k \)th widest component is processed in the \( k \)th freest cluster.

It tends to distribute the free processors as evenly as possible among the clusters.

## 5 Experimental set up

We use CSIM discrete event simulator [2] to model the system. We consider a system made up of 5 homogeneous clusters of 20 nodes each. Inter-cluster communication is not modeled. This implies that co-allocation do not affect the execution time of multi-component jobs. Jobs are rigid and online with exponentially distributed execution and inter-arrival times (mean execution time = 10). These statistical distributions have also been used in previous related work [?]. Job sizes are generated from \( D(0.85) \) on the interval \([1, 38]\).

We consider a case where \( \text{thres} = 11 \). Large jobs (top 10\%) are broken into up to 4 components. Each large job has a probability \( \frac{1}{3} \) of being broken into 2, 3 or 4 components.

We use ART as the performance metric. For scheduler instances, we use the variation ART with utilization. When studying performance sensitivity to
parameters (like depth and maxJumps), we use a mean inter-arrival time of 0.62. This generates a load of 0.811 which we consider to be high enough to show scheduler differences. All data values are recorded at a maximum ART absolute error of 0.05 at 95% confidence interval.

For group-wise performance evaluation, jobs are grouped by size into four groups $Q_1, Q_2, Q_3$ and $Q_4$. The boundaries for the groups are the size lower quartile, median, and upper quartile. Groups $Q_1, Q_2, Q_3$ and $Q_4$ consists of jobs where $size = 1, 2 \leq size \leq 3, 4 \leq size \leq 7$ and $8 \leq size \leq 38$ respectively. Their respective job composition is 24.5%, 26.4%, 25.7% and 23.4% and the respective load composition is 5.2%, 12.0%, 24.4% and 58.4%.

6 Greedy vs FPFS for Selected Parameter Instances

In this Section, we compare the performance of FPFS(5), Greedy(5,5) and Greedy(5,20). The parameter choices are to help us compare performance at the same maxJumps but different depth values.

Figure 1 and Figure 2 show entire job stream and job groups’ performance variations. For the entire job stream, Greedy(5,20) performs best while Greedy(5,5) performs worst. For $Q_1$ and $Q_2$, FPFS(5) performs best and Greedy(5,5) performs worst. For $Q_3$, Greedy(5,20) performs better than Greedy(5,5) and FPFS(5) which perform equally. For $Q_4$, FPFS(5) performs worst while Greedy(5,20) performs best.

Overall, FPFS outperforms the greedy scheduler for small jobs and the greedy scheduler outperforms FPFS for large jobs. There is a smaller performance deviation among jobs for the greedy scheduler compared to FPFS. The greedy scheduler is therefore fairer than FPFS.
Fig. 2. Performance of FPFS(5), Greedy(5,5) and Greedy(5,20) for $Q_1$ (top left), $Q_2$ (top right), $Q_3$ (bottom left) and $Q_4$ (bottom right)

Since the greedy scheduler gives the scheduling chances to large jobs first, cases where large jobs delay starting of processing due fragmentation by small jobs are reduced. For the greedy scheduler, large jobs get better performance compared to FPFS. The reverse, however, is true for small jobs. This is because FPFS, unlike the greedy scheduler, can fish small jobs from deep in the queue. Since the depth parameter limits the greedy scheduler sample space, a very low value of it leads to a poor performance. If depth is large enough, the better packing scheme of the greedy scheduler leads to performance benefits.

The group-wise performance trend shows that the choice of a better scheduler is not straightforward. If we are to compare Greedy(5,5) with FPFS(5), Figure 1 show us that FPFS(5) outperforms Greedy(5,5). From Figure 2, we observe that FPFS(5) outperforms Greedy(5,5) for $Q_1$ and $Q_2$ (50.88% of the jobs, 17% of load), they have (approximately) similar performance for $Q_3$ and Greedy(5,5) outperforms FPFS(5) for $Q_4$ (23.44% of the jobs, 58% of load). The ART value shows that FPFS(5) outperforms Greedy(5,5) though this is true for less than a fifth of the load.
7 Variation of Scheduler Performance with Parameters

7.1 Performance variation with $maxJumps$

To study the variation of performance with $maxJumps$, we use two $depth$ values (5 and 20) and vary $maxJumps$. From Figure 3 and Figure 4, we observe that (i) the performance when $maxJumps = 0$ is the same, (ii) there is an overall improvement in performance as $maxJumps$ increases, (iii) there is a high rate of performance improvement at low $maxJumps$ values, (iv) FPFS performs best for small jobs and (v) the greedy scheduler, when $depth = 20$ performs best for large jobs.

When $maxJumps = 0$, FPFS and the greedy scheduler are equivalent to FCFS hence similar performance. The poor performance of FCFS is mainly because of large jobs blocking small jobs leading to capacity loss. Allowing small jobs to jump and utilize the free processors leads to better performance. Since FPFS have no limit on how deep into the queue small jobs can be picked, small jobs register very good performance for FPFS. Small jobs however have an effect of fragmenting the system to the disadvantage of large jobs. Small jobs perform poorer for the greedy scheduler due to the limitation of depth. The greedy scheduler however employs a better packing scheme which reduces the extent of fragmentation. This is advantageous to large jobs which explain why they register better performance compared to FPFS.

7.2 Performance variation with $depth$

We use two $maxJumps$ values (5 and 20) to study the variation of performance with $depth$. From Figure 5 and Figure 6, we observe that (i) increasing $depth$
Fig. 4. Performance variations of FPFS and Greedy where depth = 5 and depth = 20 for job groups $Q_1$ (top left), $Q_2$ (top right), $Q_3$ (bottom left) and $Q_4$ (bottom right).

increases performance, (ii) there exists a value of depth beyond which the greedy scheduler outperforms FPFS, (iii) the depth in (ii) is lower for large jobs and low maxJumps values but high for small jobs and high maxJumps values and (iv) there is a minimal improvement in performance when depth is high.

Since increasing depth provides the greedy scheduler with more jobs to pick from, capacity loss is lowered leading to improved performance. If depth is high enough, there is a high possibility that a job to be scheduled will be found. Increasing depth further creates minimal impact. Since the value of depth restricts the accessibility of small jobs deep in the queue, FPFS outperforms the greedy scheduler for small jobs.

8 Conclusion and Future Work

We have shown that the greedy scheduler performs better than FPFS for large jobs while FPFS outperform the greedy scheduler for small jobs. The values of depth and maxJumps need to be large enough to get good performance. We have also observed that the greedy scheduler is fairer than FPFS.

Our work also opens up some future research. There is need to investigate the effect of communication on the relative performance of the schedulers, the effect
Fig. 5. Performance variations of FPFS and Greedy for maxJumps values of 5 and 20 using the entire job stream

of other parameters like thres as well as investigating alternative representation of the hardness function. More research work can also be done on cases where some or all jobs are moldable.

References

1. The Top 500 Supercomputer sites. [Accessed, 17th March, 2007]
Fig. 6. Performance variations of FPFS and Greedy with depth for maxJumps values of 5 and 10 for $Q_1$ (top left), $Q_2$ (top right), $Q_3$ (bottom left) and $Q_4$ (bottom right).