Computer Cluster Scheduling Algorithm Based on Time Bounded Dynamic Programming

I.Grudenic and N. Bogunovic
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of electrical engineering and computing, Zagreb, Croatia
{igor.grudenic, nikola.bogunovic}@fer.hr

Abstract - Computer clusters are currently the most used distributed computer architecture. Efficient utilization of computer cluster depends on a scheduling policy that is applied. Scheduling of jobs in computer cluster is a complicated task due to frequent changes in the workload. In this paper we present scheduling algorithm that is based on EASY backfilling scheduling strategy. Dynamic programming with time restriction is used to calculate as good schedule as possible within given time constraints. Algorithm is evaluated on several computer cluster workloads and is shown to outperform original backfilling strategy.

I. INTRODUCTION

Computer clusters are among most used general purpose distributed computer architecture [1]. They also constitute up to 82% of top 500 supercomputers [2]. Other 18% of supercomputers are massively parallel processors and constellations.

Efficient usage of the computer cluster can be achieved by enforcement of appropriate usage policies and by the choice of the scheduling algorithm. Scheduling in parallel systems is at least NP hard [3]. Reason for additional complexity is uncertainty of future events such as arrival of new jobs, job failures, system failures and early or late job completion. Additionally, cluster scheduler decision rate must match event arrival rate in order to utilize resources as fast as possible.

Since optimal solutions for scheduling problem cannot be found in a feasible on time manner, different scheduling heuristics are used. These heuristics target several optimization goals like fairness, system utilization and average system response. Achieving optimization goals must be made avoiding user and job starvation while keeping decision rates as high as possible.

In this paper we propose scheduling strategy for management of non-preemptive rigid parallel jobs [4] in a homogenous computer cluster. Proposed strategy is based on the EASY backfilling algorithm [5]. It relies on EASY mechanism for avoidance of job starvation and optimizes on current resource utilization at every decision making time. Optimization is done by employing exhaustive search of the scheduling possibilities using dynamic programming technique [6]. Since exhaustive search on a large solution space is time consuming, algorithm search space can be constrained with configurable time limit.

This paper is organized as follows. Section II contains survey short survey of cluster scheduling algorithms. Section III introduces the proposed scheduling algorithm. Performance of the proposed algorithm is analyzed in section IV.

II. RELATED WORK

Performance of scheduling in a computer cluster can be measured in a variety of manners. Users typically prefer system that is fair and offers as high average response as possible. Organizations prefer highly utilized systems for economic reasons with lesser desire for fairness.

Due to somewhat conflicting goals, uncertain environment and time constraints cluster schedulers are designed to work on a slightly reduced version of target goals. Input for the cluster scheduler is consisted of a prioritized waiting job queue, set of jobs currently running and a set of resources. Priorities in the waiting job queue are used to abstract away fairness and other policies in the system. Schedulers are not obliged to strictly follow defined priorities because this would result in inefficient schedules, but some method of starvation control must be employed.

Response of the system is usually expressed as average job wait time and average job slowdown [7] that are defined as:

$$Average\ slowdown = \frac{\sum_{j \in Jobs} Max(\frac{W(j)+R(j)}{Max(R(j),\tau)}, 1)}{|Jobs|}$$

$$Average\ wait\ time = \frac{\sum_{j \in Jobs} W(p)}{|Jobs|}$$

where $W(j)$ and $R(j)$ denote job wait time and runtime for the job j. Average job slowdown is usually calculated by replacing runtime of very short jobs by predefined constant τ. This is done in to avoid schedulers prioritizing short jobs in order to improve slowdown metric.

Simple heuristics based on reordering of the waiting job queue priorities according to job runtime and parallelization level [8] are sometimes used as a theoretical base for comparison of algorithms. Shortest job first algorithm (SJF) is proven to produce schedule with

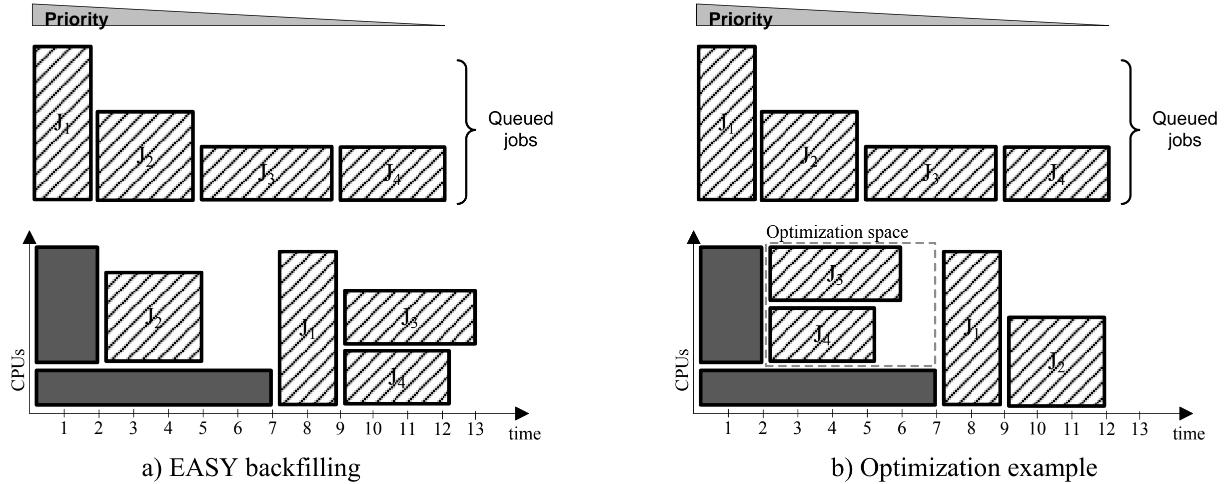

Fig. 1. Motivation for schedule optimization

minimal average job slowdown but is completely disregarding custom priorities and has no starvation control.

Major cluster management systems employ different types and modifications of backfilling based algorithms. Backfilling algorithms create reservations for subset or all the jobs in the system and allow jobs to run earlier as long as reservations in the system are not prolonged. EASY backfilling [9] that creates reservation only for the top priority job and conservative backfilling that create reservation for all the jobs in the system are two varieties that are most widely used.

Variant of selective reservation backfilling [10], where reservation is made only for jobs that have been waiting longer then predefined time constraint, is shown to reduce average slowdown by up to 70%. This reduction is obtained by implicit reversal of given job priorities. Method for dynamic partitioning of computer cluster [11] and scheduling partitions using backfilling algorithm is shown to reduce average slowdown by up to 70%. Jobs are assigned to partitions according to their expected runtime. Algorithm that creates reservations depending on the order obtained by analyzing subset of the waiting jobs [12] is shown to decrease average job slowdown for up to 30%.

Genetic algorithms are usually used for static scheduling problems where computation speed is not an issue. Input bounded genetic algorithm [13] is proposed to cope with scheduler volatile environment. This method is shown to decrease makespan of the system up to 50% when compared to round robin and lightest load first heuristics.

There are other algorithms for computer cluster scheduling but they either require additional information from the user or work with job types that are not rigid or non-preemptive.

III. TIME BOUNDED SCHEDULING ALGORITHM BASED ON DYNAMIC PROGRAMMING (DPSA)

Time bounded scheduling algorithm (DPSA) that is described in this section is based on EASY backfilling strategy with further optimization in 'hole' filling. Optimization is performed by exhaustive search that is interrupted when time limit expires. Most efficient solution found within given time constraint is used. Several varieties of the algorithm are proposed that differ in order in which exhaustive search is performed and which favor different types of optimization solutions. Subsection A describes motivation behind this type of optimization approach, while subsection B introduces the algorithm. Different variations of DPSA algorithm obtained by applying different job prioritization schemes are given in subsection C.

A. *Motivation*

EASY backfiling algorithm schedules jobs by creating reservation for the top priority job and allows other jobs to run sooner only if such decision doesn't prolong defined reservation. Other jobs are scanned in descending priority order and job is executed if there are enough available resources at the moment of the decision. This strict priority search can result in underutilization of resources and less efficient schedules. Left side of the Fig. 1 shows an example of EASY backfilling. At time=0 there are two jobs running in the system and four jobs waiting for execution. Jobs are sorted according to priorities with J_1 having the highest priority. At time=2 one of the running jobs is completed and EASY decision process is invoked.

EASY starts by creating reservation for the top priority job J_1 at time=7. This is followed by inspecting J_2 and starting it immediately (at time=2) since there are enough available resources. Scanning of the waiting queue continues, but there are is a lack of resources for either J_3 or J_4. Next decision point occurs at time=7 when J_1 is started and reservation for J_3 is made. Jobs J_3 and J_4 are started at time=9 which and this is the last scheduling decision made by EASY at that exact moment.

It can be observed that more efficient schedule such as one in the right side of the Fig. 1 is possible. The schedule on the right side improves on makespan, system utilization and average job slowdown. At time=2 EASY created reservation for J_1 and decided to run job J_2 just because it has the highest priority and can be executed immediately. In our approach we do keep the reservation for the highest priority job because this ensures algorithm to be starvation

free. When reservation for J1 is made optimization space (Fig. 1) becomes available for scheduling. Jobs can be combined in the optimization space in different ways and complexity of the problem can grow exponentially with size of this space.

In order to reduce the complexity it is decided to optimize only for current utilization and to focus on jobs that can be started immediately. Problem of optimization that corresponds to two dimensional packing of jobs is simplified to finding a subset of jobs that can be executed momentarily and which employ as much resources as possible. Construction of more complex schedules that include plans for starting jobs in the future is possible, but it is both time consuming and can be imprecise. Inaccuracy of complex schedules is usually caused by poor user runtime estimations [14].

Application of exhaustive search for example in Fig. 1 results in running J_3 and J_4 in parallel instead of J_2 since combination (J_3,J_4) maximizes utilization at time=2. This results in greater overall utilization, shorter makespan and reduction in average system slowdown.

B. *DPSA Algorithm*

DPSA Algorithm is designed to create reservation for the top job and perform an exhaustive search on available jobs to find the subset that maximizes utilization of free resources. Algorithm is invoked when either new job enters the system or one of the running jobs completes its execution. This is easily extended to scenarios in which users are allowed to cancel jobs and where resources may malfunction.

Outline of the algorithm that is consisted of the main DPSA procedure and procedure for resource utilization optimization is presented in Fig 2. Main DPSA procedure begins with clearing old reservations and starting high priority jobs that can be executed immediately (line 3-7). This is followed by creating reservation for the top priority waiting job that cannot be started due to lack of available resources (line 8). Call to the *OptimizeUtilization* procedure is preceded by initialization of free resource list *lr* and eligible job list *lp*. Eligible jobs are the ones that require less then or all the free resources.

Procedure *OptimizeUtilization* is a recursive method with iterations dedicated to adding only one job to the solution being constructed. The procedure has several arguments including current job index, list of eligible jobs, list of free resources and it dynamically produces solutions that are stored in the parameter *tempSubset* during the construction. Solution contains set of pairs (j_i,rs_i) which denotes that job j_i should be executed on set of resources rs_i. The best solution found during the execution of the algorithm is stored into *topJobSet* parameter. Optimization process starts with comparison of best solution that is found yet with the solution currently being constructed. If the solution in the construction tops best solution's utilization it becomes the new best solution (lines 16-18).

Since *OptimizeUtilization* should perform exhaustive search on a solution space, every iteration of the procedure

```
1  DPSA(){
2    ClearReservations()
3    p0=top priority waiting job
4    while(AvailableResources(p0)){
5      StartJobs(p0)
6      p0=top priority waiting job
7    }
8    CreateReservation(p0)
9    lr=list of currently free resources
10   lp=list of jobs in waiting queue that need no more than |lr| resources (excluding p0)
11   OptimizeUtilization(1,lp,lr,{},topJobSubset)
12   StartJobs(topJobSubset)
13 }
14
15 OptimizeUtilization(jobIndex,lp,lr,tempSubset,topJobSubset)
16   if(Utilization(tempSubset)>Utilization(topJobSubset)){
17     topJobSubset=tempSubset
18   }
19   i=jobIndex
20   while(i<|lp|){
21     feasibleResources={r|(r∈lr)∧(Availability(r)>=Runtime(lp(i)))}
22     if(ResourceConsumption(lp(i))<=|feasibleResources|{
23       dedicatedResources= subset of feasibleResources that minimizes on resource availability
24       tempSubset.Add(lp(i),dedicatedResources)
25       OptimizeUtilization(i+1,lp,lr-dedicatedResources,tempSubset,topJobSubset)
26       tempSubset.Delete(lp(i),dedicatedResources)
27     }
28     i++
29   }
27 }
```

Fig. 2. DPSA algorithm

is designed to add every possible job to the solution than is being created. In order to prevent duplicating solutions every procedure iteration scans and works only with jobs that haven't been used by other iterations of the same procedure that reside higher in the call tree. This is controlled by the argument *jobIndex* that represents cardinal number of the first job that hasn't been used yet.

For every unused job set of feasible resources is computed (line 21) . Feasible resources for any job *j* are currently free resources that ether have no future reservation or future reservation do not conflict with immediate execution of job *j*. If quantity of feasible resources is less or equal to the amount required by the given job this job is added to the temporary solution. Since there may be many mappings of a particular job to feasible resources, algorithm chooses the one that contains resources with minimal availability (line 23). Chosen mapping is then added to the temporary solution (line 24) and recursive call of *OptimizeUtilization* is performed (25). This is followed by removing the job added to the temporary solution and iterating over the next job.

Time boundness of this algorithm is easily achieved by adding a time or iteration count constriction on the top of the procedure *OptimizeUtilization*. This algorithm ensures absence of job starvation by creating reservations for the highest priority job that cannot run immediately. Since there is finite number of jobs that have priority higher than any job *j* in the system, and since reservations for higher priority jobs are performed in finite time it can be deduced that job *j* will also start in finite time. However there is no possibility to determine the latest possible start of any but highest priority job.

C. *Variants of the DPSA algorithm*

It is fairly easy to determine which schedule solution to take in cases when solutions differ in current utilization of resources. Sometimes it may happen that two solutions have same utilization while competing to be enforced in the system. This can be resolved by numerous heuristics and we propose three of them that rank solutions with equal utilization according to job priorities (DPSAp) and resource usage (DPSAn and DPSAw).

DPSAp variant differentiates two solutions by comparing their highest priority jobs. The solution that is preferable is the one in which top job has a higher priority. If highest priority jobs are the same, second highest priorities are compared and process is repeated recursively. This variant is beneficial since it implicitly supports enforcement of system policies. DPSAn variant favors scheduling solutions containing jobs that use lesser amount of resources. This allows more jobs to execute on the same fixed amount of available resources which can result in decrease in average slowdown and average wait time. DPSAw variant is a direct opposite of DPSAn algorithm and it therefore favors solutions with jobs that consume greater amount of resources. The reasoning behind DPSAw procedure is to start resource intensive jobs immediately if possible in order to prevent future schedule fragmentation.

Modification to base algorithm (Fig. 2) is needed in order to perform DPSAn and DPSAw variants of the algorithm. DPSAp variant is inherently implemented. Simple modification might just add on comparison of utilizations obtained by the best found solution and solution than is most recently constructed (line 16). This is not a best practice since algorithm may be time constrained so more efficient implementation would create solutions in order proposed by DPSAn or DPSAw. Realization of time constrained efficient DPSAn and DPSAw variants is done by presorting jobs in the waiting queue by size instead of priority (line 10). Ascending order is used in DPSAn and descending in DPSAw algorithm.

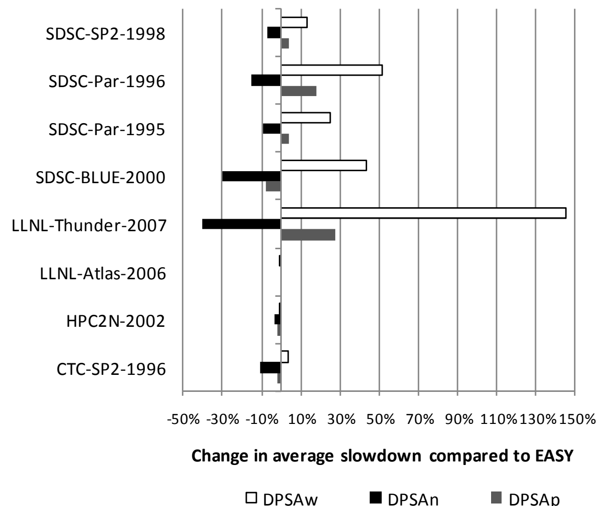

Fig. 3. DPSA performance results

IV. RESULTS

Three variants of the DPSA algorithm are tested against eight computer cluster workloads. Multiple simulations are performed using discrete time simulator [15]. Computer cluster workloads originate from Parallel workloads archive [16]. Targeted workloads differ in the number of available resources and system utilization.

Simulations are performed in order to compare efficiency of DPSA variants and original EASY backfilling algorithm. In the experiment there was no time bound imposed on the algorithm since every scheduling decision was made within few seconds. This is due to specifics of the given workloads and elimination nature of the algorithm that disregards ineligible jobs during the optimization. Time bound should be used on systems in which many options for optimization exist.

Results that show change in average slowdown of DPSA variants over EASY are presented in Fig. 3. It can be observed that DPSAw algorithm always performs worse than EASY resulting in up to 145% average slowdown increase. This leads to conclusion that raising priority of resource intensive jobs results in poor slowdown rates.

Standard DPSAp variation is shown to outperform EASY in half of the tested computers clusters where average slowdown decreased from 1% up to 8%. In other four computer clusters DPSAp resulted in slowdown increase of up to 17% at *LLNL-Thunder-2007* computer cluster. Although it may be expected that optimization on

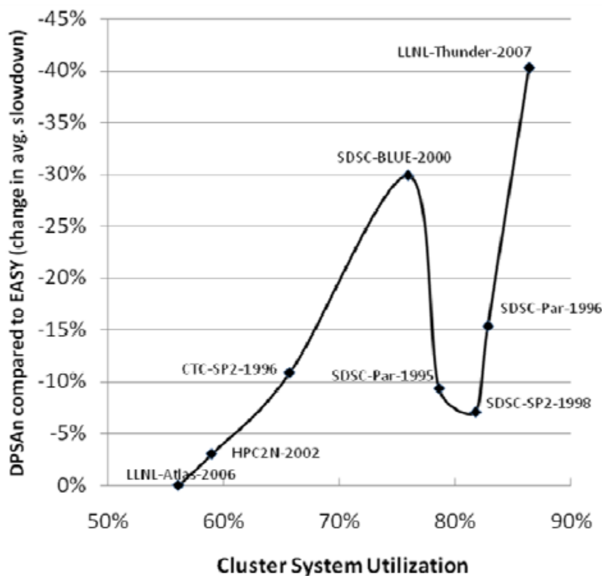

Fig. 4. DPSAn performance and utilization correlation

current utilization while respecting priorities will lead to better slowdown results it can be noted that job duration is an important factor for the targeted metric. Scheduling strategies that increase priority of short jobs are guaranteed to improve on overall slowdown, but this can hardly be achieved since exact runtimes are unknown before the job completion.

DPSAn is the only DPSA variety that outperforms EASY for all the simulated cluster workloads. Decrease in average slowdown over EASY ranges from 0,3% for *LLNL-Atlas2006* up to 40% for *LLNL-Thunder-2007* computer cluster. In order to determine the type of workload for which DPSAn will produce significant improvement performance results are correlated with utilization of the target workloads. Visual representation of the correlation is given in Fig. 4.

It can be noted that benefits of using DPSAn are negligible for systems that have low utilization such as LLNL-Atlas-2006 and HPC2N-2002. In these systems there are usually small amounts or no jobs in the waiting queue so optimization options are greatly reduced. Performance gains (reductions in average slowdown) over EASY algorithm are shown to rise as the system utilization rises. The correlation between the gains and utilization seems to be linear except for the anomaly caused by SDSC-Par-1995, SDSC-Par-1996 and SDSC-SP2-1998. It must be emphasized that performance improvements greatly depend on the opportunity to rearrange waiting jobs. In some clusters systems job patterns and current system availability show modest space for improvement.

V. CONCLUSION

In this paper we presented a novel DPSA algorithm for scheduling of rigid non-preemptive jobs on a computer cluster. Algorithm is designed by dynamic programming technique and it performs exhaustive search on possible mappings of waiting jobs to unutilized resources. In order to ensure high decision rates needed by the cluster's dynamic environment DPSA is time bounded.

The DPSA algorithm improves over EASY's strict priority backfilling scheme by allowing for higher utilization of resources at the time of decision making. Reservation system of EASY is preserved to ensure freedom of job starvation. Three variations of the DPSA algorithm are defined which favor jobs according to priorities and resource consumption.

DPSA variants performance is compared to plain EASY scheduling for eight different workloads. Results show that DPSAn variant which favors less resource intensive jobs is a winning strategy with up to 40% decrease in average slowdown. It is concluded that DPSAn variant shows greatest performance improvement for highly utilized computer clusters.

Since any optimization goal other then maximizing utilization of currently available resources strongly depends on precise job runtimes we plan to investigate on possibilities of their prediction. Finding correlation between scheduler decision time and performance of the different scheduling algorithms is also one of the possible topics in computer cluster scheduling research.

VI. REFERENCES

[1] IDC HPC Market Update, http://www.hpcadvisorycouncil.com/events/china_workshop/pdf/6_IDC.pdf

[2] TOP500 Supercomputing Sites, http://www.top500.org/

[3] E.G. Coffman Jr, M.R. Garez, D.S. Johnson, R.E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms", *SIAM Journal on Computing*,Vol. 9, p. 808, 1980.

[4] D.G. Feitelson, L. Rudolph, "Toward Convergence in Job Schedulers for Parallel Supercomputers", *In Jobs Scheduling Strategies for Parallel Processing*, *Lecture Notes in Computer Science*, Vol. 1911, p. 1, 2000

[5] D.A. Lifka, "The ANL IBM SP Scheduling System", *Lecture Notes in Computer Science*, Vol. 949, p. 303, 1995.

[6] D.B. Wagner, "Dynamic Programming",*The Mathematica Journal*, Vol. 5, Issue 4, Fall 1995,

[7] D. G. Feitelson, "Metrics for parallel job scheduling and their convergence", *Lecture Notes in Computer Science*, 2001, Vol. 2221, p. 188, 2001.

[8] O. Arndt, B. Freisleben, T. Kielmann, F.Thilo, "A comparative study of online scheduling algorithms for networks of workstations", *Cluster Computing*, Vol. 3, Issue 2, p. 95, 2000.

[9] D.A. Lifka, "The ANL IBM SP Scheduling System", *Lecture Notes in Computer Science*, Vol. 949, p. 295, 1995.

[10] S. Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan, "Selective Reservation Strategies for Backfill Job Scheduling", *Lecture Notes in Computer Science*, Vol. 2537, p. 55, 2002.

[11] B.G. Lawson, "Self-adapting backfilling scheduling for parallel systems", *In Proceedings of the International Conference on Parallel Processing*, p. 583, 2002.

[12] E. Shmueli, D.G. Feitelson, "Backfilling with lookahead to Optimize the Performance of Paralell Job Scheduling", *Lecture Notes in Computer Science*, Vol. 2862, p. 228, 2003.

[13] A.J. Page, T.J. Naughton, "Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing", *In the Proceedings of 19th IEEE International Symposium on Parallel and Distributed Processing*, 2005.

[14] C.B. Lee, Y. Schwartzman, J. Hardy, A. Snavely, "Are user runtime estimates inherently inaccurate?", *Lecture Notes in Computer Science*, Vol. 3277, p. 253, 2005.

[15] I. Grudenić, N.Bogunović, Computer Cluster and Grid Simulator, *In the Proceedings of the Joint Conferences Computers in Technical systems and Intelligent systems*, p. 44, 2009.

[16] Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload/