Parallel Backfill: Improving HPC System Performance by Scheduling Jobs in Parallel

Donald Riffel
dhriffel@ksu.edu
Kansas State University
Manhattan, Kansas, USA

Daniel Andresen
dan@ksu.edu
Kansas State University
Manhattan, Kansas, USA

Scott Hutchison
scotthutch@ksu.edu
Kansas State University
Manhattan, Kansas, USA

William Hsu
bhsu@ksu.edu
Kansas State University
Manhattan, Kansas, USA

ABSTRACT

High-performance computing (HPC) clusters are widely used as a platform for scientific and engineering research as well as a broad range of data analysis tasks. Demand for HPC resources continues to grow, necessitating more scalable systems and improved management of cluster resources. Job scheduling algorithms are key components of managing the allocation of cluster resources. A common algorithm that is used in many production systems is backfilling, which provides an efficient and feasible approach to scheduling. Many variations of backfilling have been created and studied which aim to improve its performance, but there are still opportunities in this field. In this paper, we propose a new approach named Parallel backfilling which improves scheduling throughput without increasing execution time in production environments. Our concept is to allow for multiple backfill "workers" to process the waiting job queue in parallel, increasing the rate of scheduled jobs and thus improving system turnaround time for users. We present simulated results based on job traces from the Beocat HPC cluster at Kansas State University that show significant improvement in average job wait times and scheduler throughput. We conclude that Parallel backfill provides better performance than traditional backfill and some of its variants, and compare our results with a selection of scheduling optimizations.

CCS CONCEPTS

• **Hardware** → *Testing with distributed and parallel systems*; • **Software and its engineering** → **Scheduling**; • **Theory of computation** → **Parallel algorithms**;

KEYWORDS

High-Performance Computing, Scheduling, Performance, Slurm

ACM Reference Format:
Donald Riffel, Daniel Andresen, Scott Hutchison, and William Hsu. 2024. Parallel Backfill: Improving HPC System Performance by Scheduling Jobs in Parallel. In *Practice and Experience in Advanced Research Computing (PEARC '24), July 21–25, 2024, Providence, RI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626203.3670610

PEARC '24, July 21–25, 2024, Providence, RI, USA

ACM ISBN 979-8-4007-0419-2/24/07
https://doi.org/10.1145/3626203.3670610

1 INTRODUCTION

High-performance computing (HPC) clusters are widely used as a platform for scientific and engineering research. Resource management (RM) systems are a critical component of these clusters as they are responsible for scheduling workloads that users submit as jobs. These jobs request resources such as processor cores, memory, graphics processors, and estimated amounts of runtime. RM systems schedule these resources initially with a primary scheduler that is quick to place incoming submissions into the active schedule. This is effective at creating a dense schedule upfront but as jobs complete earlier than expected gaps will appear. These gaps lead to a sparse schedule that is inefficient at utilizing resources and pushes new job start times further in the future. To address this another scheduler is used which performs backfilling [9]. Backfill scheduling has been shown to decrease the wait time for job execution and increase the density of cluster schedules by 20% [7][5]. Several RM systems including Slurm [10], Maui [5], and Sun Grid Engine [2] incorporate backfilling as a scheduler. These systems often provide administrators with configuration options to fine-tune backfill parameters, facilitating customization based on the specific characteristics and requirements of the HPC environment. The main disadvantage of this scheduler is that it is computationally expensive to perform per job in the queue, making it less effective in larger-scale clusters. Our paper focuses on enhancing the performance of an existing backfill scheduler in the widely used open-source Slurm RM system. We aim to improve backfill efficiency by creating a parallel implementation that makes use of readily available processor cores on HPC controller nodes. Experimental results and benchmarks are presented to showcase the effectiveness of our enhancements, with our experimental setup described in appendix section A. We also discuss encountered challenges, their resolutions, and potential future improvements.

1.1 Related Work

Previous efforts have focused on improving scheduling by creating better run time estimates, estimating job completion time through code analysis, or even replacing the backfill algorithm entirely. Some work has focused on modifying the strategies used in original

backfilling. One such work is Fattened backfilling which is a slight modification [3]. In Fattened backfill, short jobs are allowed to move forward in the schedule if they do not delay the first job in the queue *by more than the average waiting time of already finished jobs*. A similar work is Deviation backfilling, where short jobs are moved up if they do not delay the first job in the queue *more than the deviation between the user estimate and system prediction for the first job* [6].

Other works have taken more advanced approaches to improve backfilling ability. Several have targeted improving user runtime estimates as the optimization, with a recent focus on utilizing machine learning methods [8] [1] [12]. Some look at alternative priority schemes for considering jobs, such as Shortest Area First [4] which orders potential jobs based on their cumulative resource "area" requested. Most of these attempts have shown performance improvements, but many are targeted at specific workload types or require preferred circumstances. Our approach takes advantage of the fact that cluster systems are inherently designed to supply resources for parallel computation, and avoids reliance on special cases.

2 PARALLEL BACKFILL ALGORITHM

To implement our proposed solution, we began by examining the fundamental loop of the backfill scheduler in Slurm. The pseudocode of Slurm backfill is shown in Algorithm 2. This backfill is similar to the original method and begins by collecting the queued jobs and sorting them based on arrival time, as well as the running jobs and orders them based on expected termination. It also collects free and in-use resource information. It then iterates through the sorted job queue until either all jobs have been checked or a timeout occurs. For each iteration, initial viability checks are performed before proceeding to a schedule attempt section. If enough free resources are available that the job may start immediately, it does so and continues to the next job in the queue. Otherwise, if there are resources available at a future point in the schedule and those resources are available for at least the expected runtime of the job, a reservation for the job is placed at that period. This is the general algorithm and many other operations take place such as assessing the impact of preempting other jobs, user or partition usage limit checks, and Quality of Service (QOS) factor calculation. From this, we identified two avenues for introducing parallel computing threads to improve scheduling performance.

2.1 Parallel Backfill Workers

2.1.1 Core Backfill Loop. We first focused on parallelizing the core loop of the algorithm. The original algorithm first allocates any local variables and then checks if any limits on backfill scheduling are in effect. It then performs the preliminary work of collecting and sorting the job queue, running job list, and free node resources. Next, it proceeds into the backfill loop. State information is held and updated as the backfill loop through the job queue is done sequentially. This loop pops a pending job from the queue and checks that it is still a viable job to be scheduled, then proceeds to search for a potential reservation slot for it. If potential spots are found, a schedule attempt is made on those resources. Once it has either scheduled the job or failed to find appropriate resources for it, it moves to the next pending job in the queue. The loop continues to iterate until certain termination conditions are met, such as time limits or the queue of jobs to check has been exhausted. Our modification to this portion is to restructure the loop to be executed in parallel. Variables that previously maintained state across iterations are split into shared and thread-private variables. Instead of sequentially iterating through the job queue, several backfill "worker" threads are created to perform the scheduling. Each worker pops a job off the queue and executes the loop, with this cycle continuing until all jobs have been checked or timeouts are reached. The pseudocode of the main Parallel backfill loop is shown in Algorithm 1. A majority of the heavy computation is done within the schedule attempt section, leading to our next modifications.

Algorithm 1 Parallel Backfilling Algorithm

```
ParallelBackfill(N):
QueuedJobs ← Job Queue with requested resources
RunningJobs ← Running Jobs with resources used and expected runtime
FreeNodes ← Currently free node resources
{Setup}:
for all r RunningJobs R do
    OrderedRunningJobs ← Order r by expected termination time;
for all o OrderedRunningJobs O do
    UsageProfile ← Divide o into future periods based on terminations;
for all q QueuedJobs Q do
    OrderedQueuedJobs ← Order q by arrival time, shared variable;
{Parallel Region - To be executed by N number of workers}
PrivateRunningJobs ← Thread-private copy of UsageProfile
PrivateFreeNodes ← Thread-private copy of FreeNodes
{Schedule}:
while OrderedQueuedJobs do
    j ← pop(OrderedQueuedJobs)
    if j_requestedNodes ≤ PrivateFreeNodes then
        ReceiveUpdate(PrivateFreeNodes);
        StartImmediately(j, PrivateFreeNodes);
        BroadcastUpdate(PrivateFreeNodes);
        Continue;
    for all p PrivateRunningJobs P do
        if (j_requestedNodes ≤ p_availableNodes) & (j_estimateRuntime ≤
        p_availableTime) then
            ReceiveUpdate(PrivateRunningJobs);
            CreateReservation(j, p);
            BroadcastUpdate(PrivateRunningJobs);
            Break;
        Continue;
```

2.1.2 Schedule-Attempt Section. Parallel execution of this section is more complex as it is prone to race conditions. A scheduling attempt goes through several steps with the current job. First, it attempts to schedule the job on any currently available nodes that match its requested resources. If the job is still pending after this attempt, it then simulates the termination of jobs one at a time to determine when and where the job could start. It also tracks preemptable jobs and considers those during the search. If it finds a suitable slot in the schedule, it applies a reservation to the shared schedule. These steps require many iterations stepping through several shared lists: nodes, node resources, job lists, and cluster partitions. Slurm

Algorithm 2 Slurm Backfilling Algorithm

```
SlurmBackfill:
QueuedJobs ← Job Queue with requested resources
RunningJobs ← Running Jobs with resources used and expected runtime
FreeNodes ← Currently free node resources
{Setup}:
for all r RunningJobs R do
    OrderedRunningJobs ← Order r by expected termination time;
for all o OrderedRunningJobs O do
    UsageProfile ← Divide o into future periods based on terminations;
for all q QueuedJobs Q do
    OrderedQueuedJobs ← Order q by arrival time;
{Schedule}:
for all j OrderedQueuedJobs J do
    if j_requestedNodes ≤ FreeNodes then
        StartImmediately(j, FreeNodes);
        Continue;
    for all p UsageProfile P do
        if (j_requestedNodes ≤ p_availableNodes) & (j_estimateRuntime ≤ p_availableTime) then
            CreateReservation(j, p);
            Break;
        Continue;
```

includes access locks on many of its data structures, which limits concerns of any race conditions from our perspective but negatively impacts parallel performance. Our solution is to relax the access locks on some of the shared lists. Specifically, the currently running and scheduled job resource lists, and allow for concurrent reads but maintain write locks. This is effective as a majority of the lock claims involve shared data reads within the section, and shared data writes mostly occur when the final scheduling of a job is applied.

2.2 Managing Race Conditions

With these modifications, we are allowing each thread to read the resource and job state simultaneously. This can lead to a scheduling conflict, when one worker schedules its job, the other workers could then be using incorrect state information. To solve this, we maintain a record of the resources used for workers which have completed a job schedule attempt. Once another thread reaches the schedule attempt step with outdated resource state information and is considering these resources for scheduling, it can be rectified by comparison with the saved resource allocation data of the prior workers. This rectification step is simplified as node and resource state information is stored in bitmap data structures, which can make use of bitwise comparison. For a worker which is attempting to schedule a job using outdated state information, a bitwise *AND* function can determine resource overlaps, and those are trimmed from the worker's stored resource state information. The worker is then allowed to continue and break out of its attempt if the resources it requires are no longer available.

3 RESULTS AND ANALYSIS

The experimental performance of our proposed solution showed significant improvements to several aspects of the backfill scheduler in two tested scenarios: high-job and low-job queue traffic. High-job traffic displayed improvements in the throughput of jobs examined for backfill scheduling and scaled according to the number of worker threads used. Low-job queue traffic showed baseline improvement in the execution speed of individual backfill scheduler runs that also scaled accordingly. Particular improvements were seen in the average wait time of jobs, distribution of cluster utilization over time, and pending job queue length. To determine the performance of our method we observe several common metrics: Average Wait Time (**AWT**), Average Response Time (**ART**), and Bounded Slowdown (**BSLD**). AWT shows the average time a job waits from the time of submission until it begins execution. ART is a user-focused metric that measures the average time between job submission and completion, essentially being how long the user waits for a result response from the HPC system after they request a job run. BSLD is an extension of the Slowdown metric which is the ratio of response times to run times. Slowdown can be distorted by very short runtime requests, such that a lower-limit bound is used to prevent this distortion in BSLD.

$$\textbf{AWT} = \frac{\sum_{i=1}^{n} t_i^w}{n} \tag{1}$$

$$\textbf{ART} = \frac{\sum_{i=1}^{n} t_i^r}{n} \tag{2}$$

$$\textbf{BSLD} = \frac{\sum_{i=1}^{n} \frac{t_i^r}{max(B, t_i^e)}}{n} \tag{3}$$

where

- t_i^w is the wait time of the job (difference between start and submission time)
- t_i^e is the execution time of the job
- t_i^r is the response time of the job ($t_i^w + t_i^e$)
- B is the lower limit bound (10 minutes in our tests)
- n is the total number of jobs accepted for scheduling

Our method showed the greatest improvement on the high-job traffic dataset. This scenario stimulates heavy backfilling and quickly saturates the original scheduler where the waiting job queue grows faster than can be scheduled. Parallel backfill showed increased capacity and allowed deeper searches into the job queue while having a similar execution time as the original. This allowed for a significantly lower maximum wait queue size during the simulation runs per backfill worker. The original algorithm showed a maximum queue size of 50,000 jobs on average, while the 8-worker showed around 35,000. This also corresponded with cluster utilization being more concentrated to earlier in the simulation run. Both algorithms achieved near 100% utilization but Parallel maintained higher utilization earlier, showing more efficient use of available cluster resources. Table 3 shows according to our metrics the improvements per worker count when compared to the baseline runs on the high-job traffic dataset. On the top end, 8-worker backfilling showed an over 55% improvement in ART and 60% in BSLD.

For the low-job traffic scenario, parallel backfill showed performance improvements mainly in the individual backfill run times. Smaller job queue sizes may not saturate either algorithm, which would not cause them to break execution due to time limits. While our metrics do not show much parallel improvement, each run of the backfill loop was able to check the same amount of queued jobs

Table 1: Parallel Performance on high-job traffic dataset

Worker Count	AWT *(Hours)*	ART *(Hours)*	BSLD *(Hours)*
Original	45.89	50.61	143.71
2	34.35	39.05	108.40
4	22.94	27.63	71.65
8	17.12	21.80	52.82

Table 2: Parallel Performance on low-job traffic dataset

Worker Count	AWT *(Hours)*	ART *(Hours)*	BSLD *(Hours)*
Original	1.68	6.36	4.84
2	1.39	6.06	3.77
4	1.37	6.04	3.79
8	1.38	6.05	3.76

as the original algorithm in less time with multiple workers. This improvement is minimal on a per-backfill run basis yet would lead to greater cumulative time savings as the schedule matures. Table 2 shows the performance per worker count when compared to the baseline runs on the low-job traffic dataset.

4 CONCLUSIONS AND FUTURE WORK

In this work, we have presented a new variation of the backfill algorithm we call Parallel Backfill. Our purpose was to make use of the readily available parallel resources present in HPC systems to improve the scheduling of submitted workloads. We discussed the theory and implementation of our model and analyzed its improvements based on common metrics such as AWT and BSLD. Through simulation of a production system environment and workload, our method shows improvement in HPC system utilization, average wait times, and responsiveness.

Next steps involve implementing this backfill scheduler in a production environment to assess real-world performance. Along with this is applying the modifications to more recent versions of Slurm. Further optimizations can be made to the Parallel backfill algorithm, such as applying parallel processing to more of the scheduling calculations or creating a more robust race condition management system.

REFERENCES

[1] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. 2015. Improving Backfilling by using Machine Learning to predict Running Times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2807591.2807646

[2] Wolfgang Gentzsch. 2001. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, Brisbane, QLD, Australia, 35–36. https://doi.org/10.1109/CCGRID.2001.923173

[3] César Gómez-Martín, Miguel A. Vega-Rodríguez, and José-Luis González-Sánchez. 2016. Fattened backfilling: An improved strategy for job scheduling in parallel systems. *J. Parallel and Distrib. Comput.* 97 (Nov. 2016), 69–77. https://doi.org/10.1016/j.jpdc.2016.06.013

[4] Syed Munir Hussain Shah, Kalim Qureshi, and Haroon Rasheed. 2010. Optimal job packing, a backfill scheduling optimization for a cluster of workstations. *The Journal of Supercomputing* 54, 3 (Dec. 2010), 381–399. https://doi.org/10.1007/s11227-009-0332-3

[5] David Jackson, Quinn Snell, and Mark Clement. 2001. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 87–102. https://doi.org/10.1007/3-540-45540-X_6

[6] Thanh Hoang Le Hai, Khang Nguyen Duy, Thin Nguyen Manh, Danh Mai Hoang, and Nam Thoai. 2023. Deviation Backfilling: A Robust Backfilling Scheme for Improving the Efficiency of Job Scheduling on High Performance Computing Systems. In *2023 International Conference on Advanced Computing and Analytics (ACOMPA)*. IEEE, Da Nang City, Vietnam, 32–37. https://doi.org/10.1109/ACOMPA61072.2023.00015

[7] Sergei Leonenkov and Sergey Zhumatiy. 2015. Introducing New Backfill-based Scheduler for SLURM Resource Manager. *Procedia Computer Science* 66 (Jan. 2015), 661–669. https://doi.org/10.1016/j.procs.2015.11.075

[8] Kevin Menear, Ambarish Nag, Jordan Perr-Sauer, Monte Lunacek, Kristi Potter, and Dmitry Duplyakin. 2023. Mastering HPC Runtime Prediction: From Observing Patterns to a Methodological Approach. In *Practice and Experience in Advanced Research Computing (PEARC '23)*. Association for Computing Machinery, New York, NY, USA, 75–85. https://doi.org/10.1145/3569951.3593598

[9] A.W. Mu'alem and D.G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (June 2001), 529–543. https://doi.org/10.1109/71.932708 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[10] SchedMD. 2024. Slurm Workload Manager - Documentation. https://slurm.schedmd.com/

[11] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani. 2018. A Slurm Simulator: Implementation and Parametric Analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation (Lecture Notes in Computer Science)*, Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer International Publishing, Cham, 197–217. https://doi.org/10.1007/978-3-319-72971-8_10

[12] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. 2019. Improving HPC System Performance by Predicting Job Resources via Supervised Machine Learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (PEARC '19)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3332186.3333041

A RESEARCH METHODS

We used Slurm Simulator [11] to develop and evaluate our method. It is capable of simulating backfill performance by running the algorithm on a given trace of job submissions and for a particular cluster topology. The current official release of the simulator uses version 17.11 of Slurm, which is much older than the currently available 23.11 Slurm release. There has been many changes to Slurm in general between these versions, but after comparing the backfill algorithms of both we determined there to be minimal changes and as such our modifications should apply well to current Slurm versions. The simulator was deployed on a Centos7 instance and our target cluster to simulate is the Beocat HPC cluster at Kansas State University. The job trace used for testing is a set of 100,000 real jobs selected from a distribution that represents a variety of resource requests. This data was used to create two sample datasets that showcase different scheduling scenarios. One dataset had all jobs set to be submitted to the cluster within a one-week time frame to stimulate a heavy job queue load. Dataset two had its time frame set to be two weeks in submission times, which led to a low job queue load. To compare the scalability of the algorithm, simulations were run on each dataset with the original backfill algorithm and then with 2, 4, and 8 Parallel worker threads. For each of these individual simulation runs, 10 simulations were done and final performance metrics were sampled as the average across the batch.