Topology-Aware Scheduling on Blue Waters with Proactive Queue Scanning and Migration-Based Job Placement

Kangkang Li[1(✉)], Maciej Malawski[2], and Jarek Nabrzyski[1]

[1] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA
{kli3,naber}@nd.edu
[2] Department of Computer Science, AGH University of Science and Technology, Krakow, Poland
malawski@agh.edu.pl

Abstract. Modern HPC systems, such as Blue Waters, have multidimensional torus topologies, which make it hard to achieve a high system utilization and a high scheduling efficiency. The low system utilization is majorly caused by system fragmentation, which includes both internal fragmentation due to convex prism shape requirement, and external fragmentation resulted from contiguous allocation strategy. The low scheduling efficiency comes from using a brute force search to find the free block with a matching shape for each job, which is highly time consuming. In this paper, we address the topology-aware scheduling problem on Blue Waters, with the objective of improving system utilization and scheduling efficiency. To improve scheduling efficiency, we propose an efficient free partition detection method. To improve system utilization, we propose a job scheduling strategy with proactive queue scanning and a migration-based job placement algorithm. Through extensive simulations of modeled trace data, we demonstrate that our approach improves the system utilization.

Keywords: Topology-aware scheduling · Proactive queue scanning · Free partition detection · Migration · Job placement

1 Introduction

Many high performance computing systems use various types of multidimensional torus topologies for their interconnects. Four of the top ten supercomputers in the Top500 list (June 2016) have torus networks (one 3D, two 5D, and one 6D). They are widely used in systems such as Blue Waters (3D) [6], IBM's Blue Gene/Q (5D) [15], and Fujitsu's K computer (6D) [2]. In the Blue Waters system for instance, the network consists of X, Y, Z 3D dimensions with toroidal interconnect. Each dimension has 24 Gemini routers, making the system 24*24*24 torus interconnect structure. Each coordinate on X, Y, Z dimensions

N. Desai and W. Cirne (Eds.): JSSPP 2015/2016, LNCS 10353, pp. 217–231, 2017.
DOI: 10.1007/978-3-319-61756-5_12

is associated with a Gemini router. Each Gemini router is directly associated to two computing nodes, and is connected to its six neighbour routers along X, Y, Z dimensions (each dimension has two neighbour routers).

This torus topology influences the way jobs should be scheduled and placed in the system. For example, BlueGene allows allocating network links exclusively to the selected jobs to optimize their performance, but it can leave unused nodes within the system partitions, which leads to a lower utilization. On Blue Waters, a pre-defined Shape Table is adopted to accommodate each job's request. In order to allocate a job, the scheduler has to exhaustively search the entire system to find the free block with a matching shape, which leads to a high time complexity and a low scheduling efficiency.

In order to improve the application performance and runtime consistency, Blue Waters system adopts a contiguous allocation strategy [4,5] and a convex prsim shape is allocated to each job. This strategy degrades the system utilization. On the other hand, non-contiguous allocation strategy [3,12] can improve the system utilization, but it causes job performance to go down due to communication interference and increased latency. These reasons motivate us to investigate various topology-aware job scheduling strategies.

One key factor to low system utilization on Blue Waters is system fragmentation, which includes both internal and external fragmentation. The internal fragmentation results from the convex prism shape allocation, which allocates more nodes to a job than it needs. The external fragmentation, on the other hand, is caused by contiguous allocation strategy, which separates free system resources into smaller, non-contiguous blocks interspersed by allocated resources. This leads to the situation when sufficient number of free nodes cannot be contiguously allocated for a job. In this paper, we focus on developing efficient job scheduling strategies to reduce system fragmentation, hoping that it can improve system utilization.

Blue Waters system is using Adaptive Computing's Moab scheduler as system scheduler [6]. The scheduler is in charge of assigning each waiting job a priority and placing waiting jobs into the system. The ordered priority regulates the schedule order for waiting jobs in the queue and determines which jobs to select and when to allocate the selected jobs. In this paper, without loss of generality, we assume the queue is *never empty* and is already *ordered* by assigned priority. Each job is characterized by its own resource demand (number of nodes), and estimated walltime. The objective is to design an efficient job scheduling strategy to achieve a high system utilization and a high scheduling efficiency. Meanwhile, we must preserve the performance of jobs and avoid communication interference. Therefore, following the suggestions of system administrators of Blue Waters, we still need to maintain contiguous allocation strategy and allocate convex prism shape for input jobs.

The paper is organized as follows: In Sect. 2, we propose a scheduling strategy with proactive queue and system scanning. In Sect. 3, we present the free partition detection method and the multiple knapsack model for the job placement problem. In Sect. 4, we present our migration-based job placement algorithm for

solving the multiple knapsack problem. In Sect. 5, we conduct simulations to validate the efficiency of our approach. The related work is discussed in Sect. 6 and we give our conclusions and future work in Sect. 7.

2 Scheduling with Proactive Queue and System Scanning

In this section, we present a scheduling strategy based on proactive queue and system scanning. In our approach, the scheduler allocates waiting jobs in the queue to the system in scheduling cycles. At each scheduling cycle, the scheduler maintains a scan window to proactively scan the queue. In the meantime, the scheduler also scans the system to detect a set of free partitions. The size of the scan window is the depth of the scanning from the head of the waiting queue, as shown in Fig. 1. This queue scanning generates a set of jobs in the scan window ordered by priority.

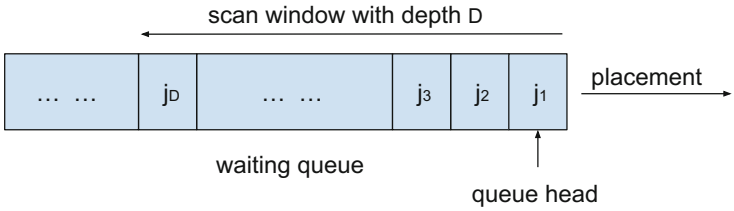

Fig. 1. Proactive queue and system scanning

The system scanning detects a set of free partitions in the system. This set of free partitions represents all available contiguous resource areas. These areas can be represented as a set of *bins*. Each bin is a 3D convex rectangular prism. Once the set of jobs in the scan window is obtained, we will group these jobs and try to place them together onto the set of free bins, all at once. This scheduling strategy has the potential to improve system utilization as multiple jobs are scheduled together, which leads to a better resource allocation. In the paper, we use the two terms (bins and partitions) interchangeably.

As described in Algorithm 1, at one scheduling cycle, starting from current queue head job, the scheduler scans the queue with depth D, and generates the set J of D waiting jobs ordered by priority. Meanwhile, the scheduler scans the system to obtain the set P of M free partitions. After that, the scheduler places each job in J (waiting job set) into P (free bin set) until all jobs in J are allocated or one job $j_i \in J$ is rejected. The detailed placement process will be discussed in Sect. 4. As each job's information (number of nodes, estimated walltime) in the scan window is known to us, this job placement process is in fact an off-line job placement.

If all jobs in J are allocated, we will wait until next scheduling cycle. Otherwise, if one job $j_i \in J$ is rejected, we will perform backfilling and place "backfilled" jobs into P. In order to implement backfilling, we need to first determine

Algorithm 1. Scheduling with Proactive System and Queue Scanning

```
1: if at one scheduling cycle then
2:    scan the queue and generate the set J of D waiting jobs: J = {j_1, ..., j_D}
3:    scan the system and generate the set P of M free partitions: P = {p_1, ..., p_M}
4:    Job Placement (J, P)                                   ▷Algorithm 5
5:    if j_i ∈ J is rejected then
6:       calculate the start time of j_i and reserve space for it
7:       perform backfilling
8:    else
9:       wait until next scheduling cycle
```

the start time of j_i and reserve space for it, which requires the following three steps:

1. Obtain the list of running jobs in the system, and sort them increasingly by their remaining completion time.
2. Starting from the time point of current scheduling cycle to the future, record the time point upon each running job's completion time, and put those time points in the timeline.
3. Go through the timeline and calculate the largest free partition in the system upon the time point of each running job's completion time. Once one sufficient largest free partition is found to accept j_i at time point t, we stop the search. Time point t is then recorded as the start time of job j_i, and the corresponding largest free partition is reserved for j_i.

With the start time t determined and space reserved, we will perform backfilling and allocate qualified backfilled jobs according to the ordered priority. The qualified backfilled jobs are those in the queue which can finish execution before the start time t of job j_i. Once P cannot accept "backfilled" jobs, we will terminate current scheduling cycle and wait until the start time of j_i. If the start time of j_i stretches multiple scheduling cycles, we will keep using "backfilled" jobs to fill in P at each scheduling cycle until the start time of j_i.

3 Free Partition Detection and Multiple Knapsack Model

As mentioned before, Blue Waters currently uses a pre-defined Shape Table to accommodate the request of each job. This Shape Table contains all topological shapes of sub-torus for job allocation. For instance, for a job with 8 node request, it corresponds to a shape of 2*2*2 in the Shape Table. In order to schedule such a job, the scheduler has to exhaustively search the entire system to find the free sub-torus block with a matching shape of 2*2*2, which is computational expensive. As an improvement, we propose an efficient free partition detection method to search the largest rectangular contiguous partition in the system.

The system is sliced into layers along the Y dimensions (X or Z dimension is also applicable), as illustrated in Fig. 2. Each dimension has side length of M

($M = 24$ in the case of Blue Waters). To obtain a maximum rectangular free block on one layer, it takes time of $O(M^2)$ through the method of construction. As for the entire 3D system, it takes another $O(M^2)$ to go through the combinations of all the layers. Therefore, it takes total $O(M^4)$ to obtain the largest free rectangular partition in the system. We can schedule multiple jobs into this partition all at once instead of just one job.

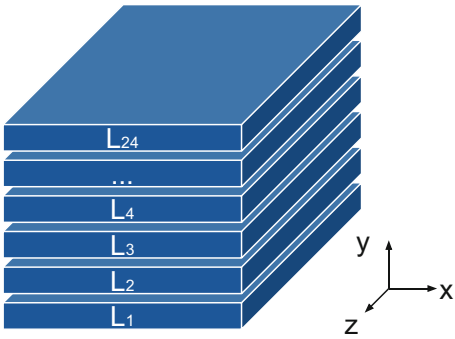

Fig. 2. Partitioning of the system into layers by Y axis

Given job shape, job placement into bins can be expressed as a 3D multiple knapsack problem. Each bin can be considered as a knapsack and input jobs are the items waiting to be put into the knapsacks. Let $J = \{j_1, j_2, ..., j_D\}$ be the set of all D waiting jobs ordered by priority in the scan window. Each job j_i has weight w_i, with profit p_i. Let $K = \{k_1, k_2, k_3, ..., k_M\}$ be the set of M knapsacks, which comes from the free bin set $P = \{p_1, p_2, p_3, ..., p_M\}$ obtained in Algorithm 1. Each knapsack k_j has capacity of C_j, which will be reduced as more jobs are placed into the knapsack. We want to find a placement for the D jobs together into the set P of free bins to maximize the total profit. The mathematical formulation is as below:

$$Max : \sum_{i=1}^{D} \sum_{j=1}^{M} x_{ij} p_i \tag{1}$$

$$Subject\ to : \sum_{j=1}^{M} x_{ij} \leq 1, \quad \forall i = 1, 2, ..., D \tag{2}$$

$$\sum_{i=1}^{D} x_{ij} w_i \leq C_j, \quad \forall j = 1, 2, ..., M \tag{3}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i = 1, 2, ..., D, \ \forall j = 1, 2, ..., M \tag{4}$$

$$C_j \geq 0, \quad \forall j = 1, 2, ..., M \tag{5}$$

$x_{ij} = 1$ means job i is put into knapsack j, and $x_{ij} = 0$ means job i is not put into knapsack j. The physical meaning of both weight w_i and profit p_i is the

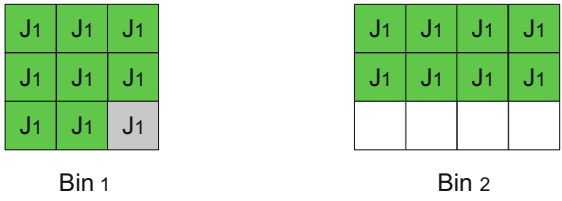

Fig. 3. Illustration of difference in internal fragmentation from job placement

job size, which is the number of requested nodes of job i. The capacity of each knapsack can never be negative, but it will be reduced as more jobs are put into this knapsack.

Based on this multiple knapsack model, maximizing the system utilization can be transformed into maximizing the objective of Eq. 1. As the sizes of input jobs and capacities of free bins are heterogeneous, this multiple knapsack problem is NP-hard and requires a heuristic algorithm, one example of which is presented in the next section.

4 Migration-Based Job Placement

In this section, we propose a migration-based job placement heuristic algorithm to solve the multiple knapsack problem. The intuition of this heuristic is to minimize the internal fragmentation brought in by the job placement process.

Once the set J of waiting jobs in the scan window and the set P of free partitions are obtained in one scheduling cycle, we will place each incoming job in J into one of the bins in P. However, the extent of internal fragmentation (the number of idle nodes due to using convex shape) is different if we place a job in different bins. Figure 3 gives a 2D example.

As shown in Fig. 3, there is an incoming job J_1 with 8 nodes request. If we place it in Bin 1, it will lead to one idle node (the grey area), as the topological layout of Bin 1 is 3*3. However, if we place it in Bin 2, it leads to no internal fragmentation, as the layout of Bin 2 is 3*4. Therefore, Bin 2 is a better choice and more preferable than Bin 1 in minimizing the internal fragmentation.

Thus, for each job, there are preference differences in placing it into different bins. Each bin is ranked by the extent of the internal fragmentation this bin can bring in. We are looking for the best bin that leads to the minimal internal fragmentation. However, if the resources in the best bin are not sufficient for an incoming job, we have two options.

1. **Direct Placement:** Among all the bins which have enough resources to accept the incoming job, we select the one with minimal internal fragmentation.
2. **Migration-based Placement:** We try to find one "victim" job on the best bin, and migrate it into another bin, as shown in Fig. 4. In that case, we can make some more room for accepting the incoming job.

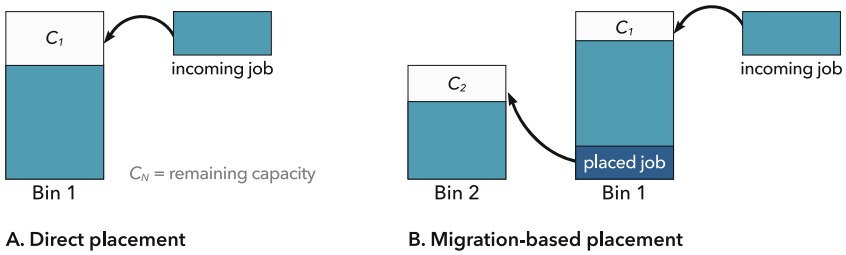

Fig. 4. An illustration of the job placement. When using direct placement, the incoming job is assigned to bin 1 with the enough remaining capacity C_1. When using migration, we can find an already placed job ("a victim") to be migrated to another available bin with enough capacity, such as bin 2 in the picture above

The first option is not optimal as it misses the opportunity to place the incoming job on the best bin, especially when the internal fragmentation on the best bin is much less than that on the other bins. Therefore, we want to take advantage of migration to do the placement. However, migration has constraint. For the migrated victim job, there might be an internal fragmentation increase due to the change of host bin. As our objective is to reduce the overall internal fragmentation from job placement, if the internal fragmentation increase of the migrated victim job is too large, migration will be meaningless. In that case, we would rather select direct placement without migration.

While implementing the migration, we need to find a qualified victim job for migration, which is not always possible. There are three conditions that a qualified victim job must meet:

1. it can make enough room to accept the incoming job.
2. it can find a new available bin with enough remaining capacity to accept the victim job itself.
3. the migration constraint must be satisfied, i.e., despite the internal fragmentation increase from the migrated victim job, the Migration-based Placement is better than Direct Placement in minimizing the internal fragmentation.

If such a qualified victim job is found in the best bin, we will choose to apply Migration-based Placement (Algorithm 2). If more than one qualified victim jobs exist in the best bin, we choose the victim job with the minimal internal fragmentation increase to migrate. However, if we cannot find a qualified victim job in the best bin, we will try to place the job in the next-best bin. If the incoming job still cannot be placed, we continue to try the next-next-best bin. This search goes on until the incoming job is placed or all the bins have been tried.

As shown in Algorithm 2, for the incoming job j_i, we first sort all the bins increasingly by the internal fragmentation of placing j_i in each bin p_j. The call for Direct Placement (Algorithm 3) returns $frag\_value$, which is the minimal value of the internal fragmentation among all the bins that are enough to accept

Algorithm 2. Migration-based Placement

Input: job j_i, the set P of M bins: $P = \{p_1, ..., p_M\}$
sort and rank each bin p_j increasingly by the extent of internal fragmentation of placing job j_i in p_j
$frag\_value$ = Direct Placement(j_i, P)
for $j = 1$ to M **do**
  if C_j is enough for accepting j_i **then**
    place j_i in p_j
  else if Migration Test (j_i, p_j, $frag\_value$) == **true then**
    perform migration and place j_i on p_j
  else
    continue

Algorithm 3. Direct Placement

Input: job j_i, the set P of M bins: $P = \{p_1, ..., p_M\}$
sort and rank each bin p_j increasingly by the extent of internal fragmentation of placing job j_i in p_j
for $j = 1$ to M **do**
  if C_j is enough for accepting j_i **then**
    return internal fragmentation of placing j_i in p_j
  else
    continue

j_i. After that, starting from the first bin on the sorted list (the best bin), we try each p_j to place the job j_i in it. If p_j is enough for accepting j_i, we just directly place j_i in p_j. Otherwise, we use $frag\_value$ to test the migration constraint (in Algorithm 4). If migration constraint is satisfied and a qualified victim job is found, we then perform migration and place j_i in p_j.

In Algorithm 4, first, for each already placed job j_k on p_j that can make enough space for the incoming job j_i, we try to find j_k a new best available bin, which is the one that has enough resources for j_k and leads to the minimal internal fragmentation increase among all the bins (except p_j). After that, we test migration constraint. If migration constraint is satisfied, we then mark j_k as a qualified victim job. If more than one qualified victim jobs exist, we select the best victim job on bin p_j which has the minimal internal fragmentation increase. Notably, the migration here is one-hop migration, which means that we only consider the migration of the victim job caused by the incoming job. The re-placement of victim job will not trigger another migration.

With D jobs and M bins in one scheduling cycle, assuming the average number of already placed jobs on a bin is K, the time complexity of Algorithm 4 is $O(KM)$, which is no more than $O(D)$. With one loop, the time complexity of Algorithm 3 is $O(M)$. Therefore, the total complexity of Algorithm 2 is $O(MKM) + O(M)$, which is no more than $O(MD)$ and pretty efficient.

The overall job placement algorithm presents in Algorithm 5 above, which corresponds to line 4 of Algorithm 1. The input is the set J of D waiting jobs in

Algorithm 4. Migration Test

```
Input: job j_i, bin p_j, frag_value
for each placed job j_k on bin p_j do
    if j_k's migration save enough space for j_i then
        for all the bins (except p_j) do
            find j_k a new best available bin (except p_j)
        calculate the internal fragmentation increase of migrating job j_k
        test migration constraint using frag_value
        if migration constraint is satisfied then
            mark job j_k as a qualified victim job
if one or more than one victim job exist on p_j then
    select the best victim job on bin p_j which has minimal internal fragmentation increase
    return true
else
    return false
```

Algorithm 5. Job Placement

```
Input: the set P of M bins: P = {p_1, ..., p_M}
       the set J of D waiting jobs: J = {j_1, ..., j_D}
for i = 1 to D do
    Migration-based Placement (j_i, P)                    ▷ Algorithm 2
    if j_i cannot be placed then
        reject j_i
        break
```

the scan window and the set P of free bins in one scheduling cycle. For each job $j_i \in J$, we apply Migration-based Placement algorithm (Algorithm 2) to place it into the set P of free partitions. When it comes to a job j_i that cannot be placed into P, we reject it and terminate this placement process.

As mentioned before, all jobs in the scan window are known to us, therefore, this job placement process is in fact an off-line job placement, where migration is an emulated process with no migration overhead. As time complexity of Algorithm 2 is $O(MD)$, the total time complexity of Algorithm 5 is $O(MD^2)$ with D input jobs, which is very efficient.

5 Performance Evaluation

In this section, we conduct simulations to evaluate our approach of improving system utilization. According to the information from administrators of Blue Waters, the current scheduling policy they use only achieves a system utilization of around 50% to 60%. We will show that our approach can significantly improve that utilization value.

The evaluation is performed using Blue Waters traces. For simplicity and without loss of generality, we have used Blue Waters trace model, preserving the

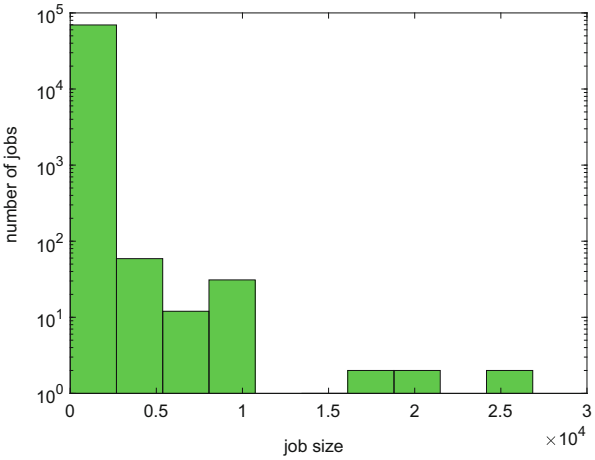

Fig. 5. Job size distribution

trace characteristics. Based on the study of trace data, we found that the largest job can almost occupy the entire system's capacity (only a few jobs like this). The minimal size job is single node job, which constitutes around 70% of the entire trace workload.

As convention, jobs with node request more than 3000 nodes are classified as extra-large jobs, and jobs with node request between 1000 to 3000 are large jobs. Jobs with node request between 100 to 1000 are medium jobs, and jobs with node request below 100 nodes are small jobs. The extra-large jobs can cause the system to drain for a long time until enough space is available to place such a large job. This drainage brings the system utilization down for a long time. To deal with these extra-large jobs, reducing system fragmentation is not enough as extra-large jobs can require half or more of system's capacity. Therefore, other approaches such as relaxing priority order are necessary to deal with extra-large jobs.

We focus on input workload that consists of small, medium and large jobs for our simulation, which constitutes 99.8% of the entire trace workload. Even if there are a few extra-large jobs, most jobs are below 3000 nodes, as shown in Fig. 5. Similarly, we also present the distribution of job walltime throughout the trace, as shown in Fig. 6. Although the dominant jobs are short, mid-length and long jobs are taken into account as well.

Using random initial input, we start with the system around half occupied. The simulation input workload has 2000 jobs, which is the approximate number of new job submissions in one day. The scheduling cycle is set as 15 min. That is, one iteration of scheduling repeats every 15 min. We allocate jobs and record system utilization at each scheduling cycle (every 15 min). As the total input workload has 2000 jobs, it requires many scheduling iterations to complete the allocation of all input jobs.

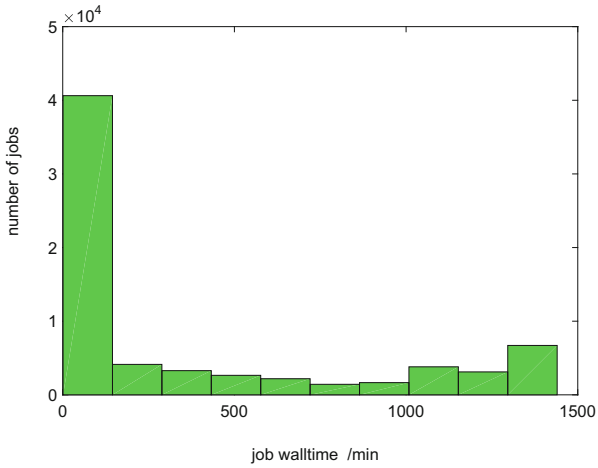

Fig. 6. Job walltime distribution

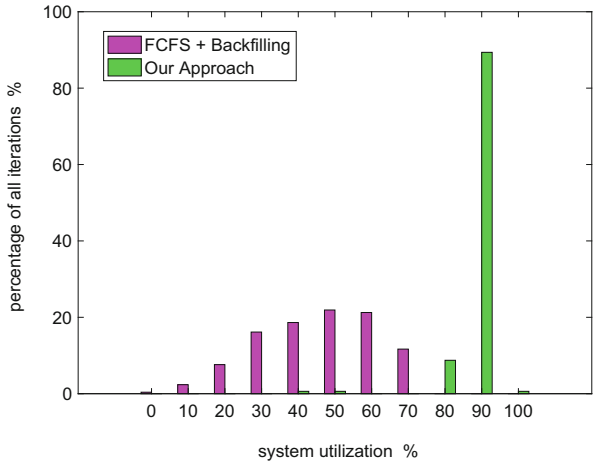

Fig. 7. Histogram of our approach with scan window size of 1000

Moreover, to prevent the scenario that the system is in low utilization because there are not enough "backfilled" jobs to be used for fully utilizing the system's capacity, we also have another set of "backfilled" jobs besides the input workload of 2000 jobs. This set of "backfilled" jobs are used for providing sufficient "backfilled" jobs to maintain system utilization. This setting is practical as the waiting queue usually has plenty of jobs for allocation.

We conduct three groups of simulations to find out the impact of scan window size on the performance of our approach, which includes a scheduling strategy using proactive queue scanning and a migration-based job placement algorithm.

In Figs. 7 and 8, the scan window size are 1000 and 500, respectively. We can see that, in most time of the scheduling iterations, our approach achieves

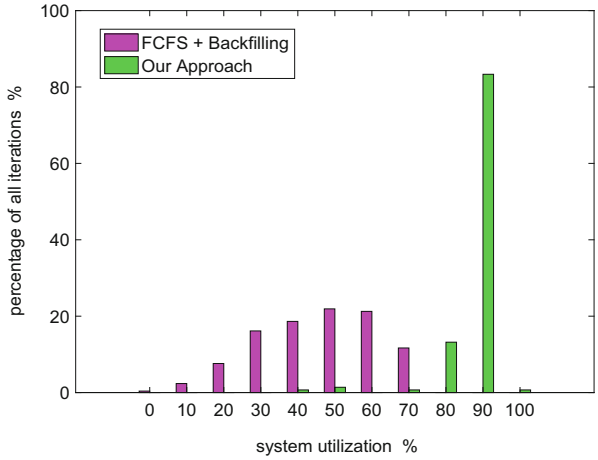

Fig. 8. Histogram of our approach with scan window size of 500

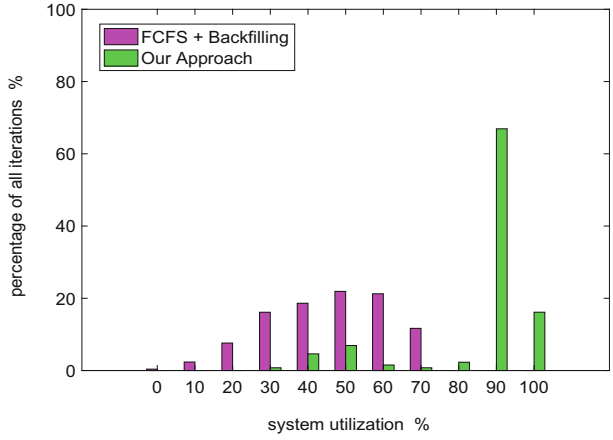

Fig. 9. Histogram of our approach with scan window size of 100

a system utilization on the level of around 90%. On the other hand, the FCFS + Backfilling strategy leads to the utilization on the level of around 40% to 70%. This shows that, our approach can improve the system utilization greatly, compared to both FCFS + Backfilling strategy and current strategy used on Blue Waters.

However, when the scan window size is 100, there are some scheduling iterations where the utilization is down to around 40% to 50%, as shown in Fig. 9. This is due to the fact that the window size is small and the free bins in the system are not fully filled. Based on this, we conclude that, large window size leads to a better system utilization. However, despite these short periods of low utilization, in most time of scheduling cycles, our approach still maintains a

system utilization of around 90% when scan window size is 100, which outperforms FCFS + Backfilling strategy and current strategy used on Blue Waters. The source code of our approach is open to community [1].

6 Related Work

To improve the task placement of applications with 2D, 3D and 4D Cartesian topologies and nearest-neighbor communication, a Topaware tool can be used [7]. There are tools such as Caypat for profiling an MPI application and to detect Cartesian grid communication patterns. This information can be used to provide runtime mapping of processes to the computing nodes using MPICH node ordering. The Topaware method requires the user to specify the required number of nodes along each torus dimension and finds the ordering by allocating nodes on subsequent XZ planes, taking into account the gaps resulting from service nodes. For mapping the 2D virtual topology to the 3D torus, a folding method is used. Topaware was evaluated using the WRF, VPIC, S3D and MILC applications.

An overview process of mapping techniques and algorithms [13] for HPC systems is presented in [8]. It discusses algorithmic strategies for topology mapping, such as graph partitioning, mapping enforcement techniques (resource binding and rank reordering), as well as existing solutions and their implementations. This provides a formal definition of the mapping as an optimization problem, and discusses the metrics such as dilation or congestion.

One of the reasons for system fragmentation lies in the discrepancy in job length/execution time, which leads to a irregular hole/fragmentation between neighbouring jobs with different finish time. In order to tackle this type of fragmentation, a walltime-aware scheduling strategy is designed in [14], which packs jobs with similar length and places them near to each other. In particular, two algorithms are developed: similar-length allocation and paired job filling. The similar length allocation algorithm tries to match waiting jobs with running jobs that share similar completion time. The paired job filling algorithm selects two jobs with the same size and similar length from the queue and schedules both jobs together. Notably, paired job filling algorithm is similar to our scheduling strategy, where multiple waiting jobs in the queue are grouped and placed together to reduce potential fragmentation.

Migration is an efficient resource management tool, which has been discussed in [9–11]. In [9], the authors present the analysis and application of scheduling algorithms that augment a baseline first come first serve (FCFS) scheduler. The author presents simulation results for migration and backfilling techniques on BlueGene/L. These techniques are explored individually and jointly to determine their impact on the system. An efficient Projection Of Partitions (POP) algorithm for determining the size of the largest free rectangular partition in a toroidal system is developed. The results demonstrate that migration may be effective for a pure FCFS scheduler, but that backfilling produces even more benefits. It is also shown that migration may be combined with backfilling to produce more opportunities to better utilize a parallel machine.

7 Conclusions and Future Work

In this paper, we addressed the problem of improving system utilization and scheduling efficiency on Blue Waters system that uses a 3D torus topology. To improve the scheduling efficiency, we propose an efficient free partition detection method. To improve the system utilization, we first propose a job scheduling strategy based on proactive queue and system scanning. After that, we model the job placement problem into a multiple knapsack model and design a migration-based job placement algorithm to give a heuristic solution. The simulations of modeled trace data demonstrate that our approach works well in terms of improving system utilization. In our future work, we will extend our study on reducing system fragmentation and improving system utilization. In particular, we will focus on improving system's capability to directly accept the incoming large and extra-large jobs. We will investigate various strategies such as relaxing priority order and migration to avoid the system drainage caused by the incoming large and extra-large jobs and to maintain system utilization without the backfilling process.

References

1. https://github.com/kangkangkenli/2016-jsspp-tas
2. Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S., Shimizu, T.: The tofu interconnect. In: 2011 IEEE 19th Annual Symposium on High Performance Interconnects, pp. 87–94. IEEE, August 2011
3. Chang, C., Mohapatra, P.: Performance improvement of allocation schemes for mesh-connected computers. J. Parallel Distrib. Comput. **52**(1), 40–68 (1998)
4. Chiu, G.-M., Chen, S.-K.: An efficient submesh allocation scheme for two-dimensional meshes with little overhead. IEEE Trans. Parallel Distrib. Syst. **10**(5), 471–486 (1999)
5. Ding, J., Bhuyan, L.N.: An adaptive submesh allocation strategy for two-dimensional mesh connected systems. In: International Conference on Parallel Processing, ICPP 1993, vol. 2, pp. 193–200, August 1993
6. Enos, J., Bauer, G., Brunner, R., Islam, S.: Topology-aware job scheduling strategies for torus networks. In: Proceedings of the Cray User Group Meeting (2014)
7. Fiedler, R., Whalen, S.: Improving task placement for applications with 2D, 3D, and 4D virtual Cartesian topologies on 3D torus networks with service nodes. In: Proceedings of Cray User's Group (2013)
8. Hoefler, T., Jeannot, E., Mercier, G.: An overview of process mapping techniques and algorithms in high-performance computing. In: Jeannot, E., Zilinskas, J. (eds.) High Performance Computing on Complex Environments, pp. 75–94. Wiley, June 2014
9. Krevat, E.: Scheduling Algorithms to Improve Utilization in Toroidal Interconnected Systems. Ph.D. thesis (2003)
10. Li, K., Zheng, H., Wu, J.: Migration-based virtual machine placement in cloud systems. In: 2013 IEEE 2nd International Conference on Cloud Networking (CloudNet), pp. 83–90. IEEE (2013)
11. Li, K., Zheng, H., Jie, W., Xiaojiang, D.: Virtual machine placement in cloud systems through migration process. Int. J. Parallel Emergent Distrib. Syst. **30**(5), 393–410 (2015)

12. Lo, V., Windisch, K.J., Liu, W., Nitzberg, B.: Noncontiguous processor allocation algorithms for mesh-connected multicomputers. IEEE Trans. Parallel Distrib. Syst. **8**(7), 712–726 (1997)
13. Mansour, N., Ponnusamy, R., Choudhary, A., Fox, G.C.: Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In: Proceedings of the 7th International Conference on Supercomputing, ICS 1993, pp. 1–10. ACM, New York (1993)
14. Tang, W., Lan, Z., Desai, N., Buettner, D., Yu, Y.: Reducing fragmentation on torus-connected supercomputers. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS 2011, pp. 828–839. IEEE Computer Society, Washington, DC (2011)
15. Zhou, Z., Yang, X., Lan, Z., Rich, P., Tang, W., Morozov, V., Desai, N.: Improving batch scheduling on blue Gene/Q by relaxing 5D torus network allocation constraints. In: IPDPS 2015 (2015)