Balancing Job Performance with System Performance via Locality-Aware Scheduling on Torus-Connected Systems

Xu Yang*, Zhou Zhou*, Wei Tang†, Xingwu Zheng‡, Jia Wang‡, Zhiling Lan*
*Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA 60616
{xyang56,zzhou1}@hawk.iit.edu, lan@iit.edu
‡Department of Electrical and Computer Engineering, Illinois Institute of Technology, Chicago, Illinois, USA 60616
xzheng18@hawk.iit.edu, jwang@ece.iit.edu
†Argonne National Laboratory, Argonne, IL, USA 60439
wtang@anl.gov

Abstract—**Torus-connected network is widely used in modern supercomputers due to its linear per node cost scaling and its competitive overall performance. Job scheduling system plays a critical role for the efficient use of supercomputers. As supercomputers continue growing in size, a fundamental problem arises:** ***how to effectively balance job performance with system performance on torus-connected machines?*** **In this work, we will present a new scheduling design named** ***window-based locality-aware scheduling*****. Our design contains three novel features. First, rather than one-by-one job scheduling, our design takes a "window" of jobs, i.e. multiple jobs, into consideration for job prioritizing and resource allocation. Second, our design maintains a list of slots to preserve node contiguity information for resource allocation. Finally, we formulate our scheduling decision making into a 0-1 Multiple Knapsack Problem and present two algorithms to solve the problem. A series of trace-based simulations using job logs collected from production supercomputers indicate that this new scheduling design has real potentials and can effectively balance job performance and system performance.**

I. INTRODUCTION

The insatiable demand in science and engineering has driven the ever-growing supercomputing systems. As the scale of supercomputers increases, so do their interconnected networks. Torus interconnection is widely used in HPC systems, such as Cray XT/XE and IBM Blue Gene series systems [2][4], due to their linear per node cost scaling and their competitive overall performance. A growing network means an increasing network diameter (i.e., the maximum distance between a pair of nodes) and a decreasing bisection bandwidth relative to the number of nodes. Consequently, applications running on torus-connected systems suffer great performance variability caused by the increasing network scale.

Job scheduling system plays a critical role for the efficient use of supercomputers. Currently two scheduling strategies are commonly used on torus-connected systems. One is so called partition based systems, where the scheduler assigns each user job a compact and contiguous set of computing nodes. IBM Blue Gene series systems fall into this category [2]. This strategy is in favor of application's performance by preserving locality of allocated nodes and reducing network contention caused by concurrently running jobs sharing network bandwidth. However, this strategy can cause internal fragmentation (when more nodes are allocated to a job than it requests) and external fragmentation(when sufficient nodes are available for a request, but they can not be allocated contiguously), therefore leading to poor system performance (e.g., low system utilization and high job response time) [11]. The other is non-contiguous allocation system, where free nodes are assigned to user job no matter whether they are contiguous or not. Cray XT/XE series systems fall into this category [4]. Non-contiguous allocation eliminates internal and external fragmentation as seen in partition-based systems, thereby leading to high system utilization. Nevertheless, it introduces other problems such as scattering application processes all over the system. The non-contiguous node allocation can make inter-process communication less efficient and cause network contention among concurrently running jobs [12], thereby resulting in poor job performance especially for those communication-intensive jobs.

Partition-based allocation achieves good job performance by sacrificing system performance (e.g., poor system utilization), whereas non-contiguous allocation can result in better system performance but could severely degrade performance of user jobs (e.g, prolonged wait-time and run-time). As systems continue growing in size, a fundamental problem arises: *how to effectively balance job performance with system performance on torus-connected machines?* In this work, we will present a new scheduling design combining the merits of partition based scheduling and non-contiguous scheduling for torus-connected machines.

Strictly speaking, a job scheduling system contains two parts, namely *job prioritizing* and *resource allocation*. Job prioritizing makes decision about the order in which jobs are allowed to run. The decision making is based on many factors, such as job size, job run-time, job priority, etc. Resource allocation decides a set of nodes allocated to each incoming job. Figure 1 illustrates a typical job scheduling system, where each job is retrieved from the wait queue and computing nodes are assigned one-by-one to this job. Many supercomputers use this kind of First-Come First-Serve (FCFS) scheduling policy. As we can see, every job gets out of the wait queue and free nodes are assigned to the job according to node identifiers. Often the topological characteristics and locality of system nodes are ignored. Numerically sequential nodes may be separated from each other in the space. A well-known approach to address the problem is processor ordering. Processor ordering usually uses space filling curve, such as Hilbert Curve, to map the nodes of the torus onto a 1-dimensional list to preserve locality information [23][22]. While processor ordering works well at the beginning of scheduling, job allocation and deallocation will eventually fragment this 1-dimensional list, making it less efficient as time goes by.

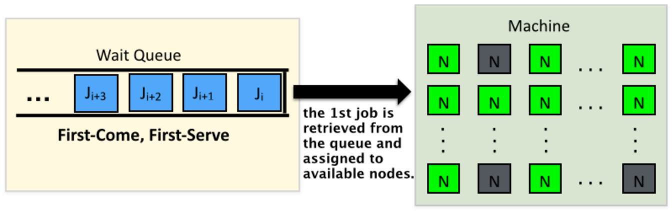
Wait Queue
...
J_{i+3}
J_{i+2}
J_{i+1}
J_i
First-Come, First-Serve
the 1st job is retrieved from the queue and assigned to available nodes.
Machine
N

Fig. 1 Typical job scheduling uses First-Come First-Serve (FCFS) scheduling policy. Jobs are removed from the wait queue and assigned with free nodes one by one. The grey squares represent busy nodes occupied by running jobs. The green squares represent free nodes.

In this paper, we present a *window-based locality-aware scheduling design*. Our design is based on two key observations. First, as shown in Figure 1, existing scheduling system makes decisions in a per-job manner. Each job is dispatched to system resources without considering subsequent jobs. While making isolating job decision may provide a good short-term optimization, it is likely to result in poor performance in the long term. Second, existing scheduling system maintains a list of free nodes for resource allocation. While special processor ordering (e.g., using a space filling curve) is often adopted for preserving node locality in the list, the node list becomes fragmented as time goes by and subsequent jobs inevitably get dispersed nodes allocation due to the lack of contiguous node list.

Rather than one-by-one job scheduling, our design takes a "window" of jobs (i.e., multiple jobs) into consideration for making prioritizing and allocation decision, to prevent the short-term decision from obfuscating future optimization. Our job prioritizing module maintains a "window" of jobs, and these jobs are placed into the window to maintain job fairness (e.g., through FCFS). Rather than allocating jobs one by one from the head of the wait queue as existing schedulers do, we make scheduling decision for a "window" of jobs at a time. Our resource allocation module takes a contiguous set of nodes as a *slot* and maintains a list of such slots. These slots have different sizes and each may accommodate one or more jobs. The allocation of the jobs in the window onto the slot list is conducted in such a way as to maximize system utilization. We formalize the allocation of a window of jobs to a list of slots as a 0-1 Multiple Knapsack Problem (MKP) and present two algorithms, namely Branch&Bound and Greedy, to solve the MKP.

We evaluate our design via extensive trace-based simulations. In this paper, we conduct a series of experiments comparing our design against the commonly used FCFS/EASY backfilling scheduling that is enhanced with processor ordering. Our preliminary results demonstrate that our design can reduce average job wait time by up to 28% and average job response time by 30%, with a slight improvement on overall system utilization.

The structure of the paper is as follows. Section II gives a detailed description of our window-based locality-aware scheduling design. Section III describes the problem formalization and two scheduling algorithms implemented in our design. Section IV–V present our evaluation methodology, trace-based simulations by comparing our design against the FCFS/EASY backfilling scheduling scheme. In Section VI, we discuss the existing work about job scheduling and allocation on HPC systems. Our conclusion is presented in Section VII.

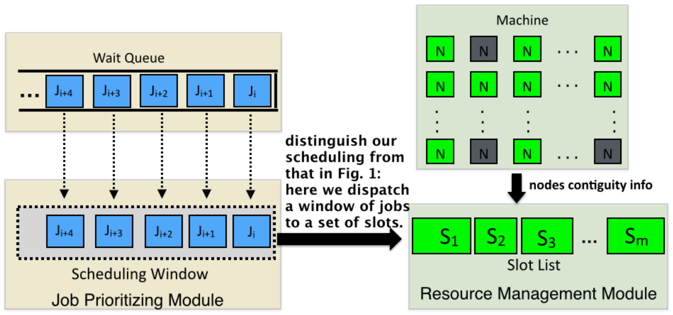

Fig. 2 Overview of our window-based locality-aware scheduling design. The job prioritizing module maintains a "window" of jobs retrieved from the wait queue, and the resource management module keeps a list of slots. Each slot represents a contiguous set of available nodes. Our scheduling design allocates a "window" of jobs to a list of slots at a time.

II. Overview

Figure 2 gives an overview of our window-based locality-aware job scheduling design. Our design contains two key parts. The job prioritizing module maintains a "window" of jobs that are retrieved from the wait queue to insure job fairness. Rather than dispatching jobs one by one as existing schedulers do, we dispatch multiple jobs at a time. Unlike existing scheduling design, the resource management module is responsible for organizing the available nodes into a set of slots. Each slot contains a contiguous set of free nodes. Here, contiguity refers to the adjacency of nodes' original positions in torus-connected system. These slots may have different sizes. A new slot appears when a job releases the nodes it was assigned. The newly released nodes merge with other free neighboring nodes to form up a new slot with growing size. A slot disappears when it (or part of it) is assigned to a job. If the job's requirement only takes part of a slot, the remaining part becomes a new slot. The slot list is updated when a job is allocated or deallocated. The resource management module needs to get feedback from the system right after a job being allocated/deallocated to update the status of the slot list. The allocation of the jobs in the window onto the slot list is conducted in such a way as to maximize system utilization.

Using window-based design can balance job fairness and system performance. In our design, rather than allocating jobs one by one from the front of the wait queue, the scheduler takes all the jobs in the window and make prioritizing decision for them at a time. The selection of jobs from wait queue to the window is based on certain scheduling rule (e.g., job arrival time in case of using FCFS policy), thereby guaranteeing job fairness. With the information of both jobs and slots, the scheduler aims to make an optimal decision in terms of allocating the jobs to the slots. In the following section, we will present our detailed scheduling strategy.

III. Scheduling Strategy

Our scheduling strategy contains two parts. We first formalize the resource allocation problem into a 0-1 Multiple Knapsack Problem (MKP), and then present two algorithms to solve the MKP.

A. MKP Formalization

We consider each slot as a knapsack and the jobs in the window are the items waiting to be put into the knapsacks. Suppose $J = \{J_1, J_2, J_3, ..., J_n\}$ is a set of n jobs in the window. Each job J_j has weight w_j, with profit p_j. And $K = \{K_1, K_2, K_3, ..., K_m\}$ is a set of m knapsacks; each knapsack K_i with the capacity of C_i. So, we want to select m disjoint subsets of jobs so that the total profit of the selected jobs is a maximum, and each subset can be put into different knapsack whose capacity is no less than the total weight of jobs in the subset. Formally,

$$\mathrm{M}ax \quad z = \sum_{i=1}^{m}\sum_{j=1}^{n} p_j \cdot x_{ij} \tag{1}$$

which is subject to the following constraints:

$$\sum_{j=1}^{n} x_{ij} \cdot w_j \leq C_i, \quad i \in \{1, 2, ..., m\} \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad j \in \{1, 2, ..., n\} \tag{3}$$

$$x_{ij} \in \{0, 1\}, \quad i \in \{1, 2, ..., m\}; j \in \{1, 2, ..., n\} \tag{4}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if job j is put into knapsack i;} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

When m=1, Multiple Knapsack problem reduces to the 0-1 knapsack problem. In our model, we can assume that

$$p_j \text{ and } C_i \text{ are positive integers,} \tag{6}$$

$$w_j \leq max_{i \in I} C_i, \quad \forall j \in \{1, 2, ..., n\}, \tag{7}$$

$$C_i \geq min_{j \in J} w_j, \quad \forall i \in \{1, 2, ..., m\}, \tag{8}$$

$$\sum_{j=1}^{n} w_j > C_i, \quad \forall i \in \{1, 2, ..., m\}, \tag{9}$$

In our model, it is guaranteed that assumption 6 can't be violated since the definitions of job's profit and weight in our maximization problem are both its size. And the capacity of knapsack is the size of the slots, which can never be negative. If there is a job *j* violating assumption 7, which means it requires too many nodes to be accommodated in any knapsack, it will be hold until a slot appears which contains enough free nodes. In our experiments, we found that this would only prolong the big jobs' wait time less than 10%. If a knapsack violates assumption 8, then it will be taken as system fragmentation. Finally, observe that if $m > n$ then the $(m - n)$ knapsacks of smallest capacity will not be included in this formalization.

The window size should be set based on the system's workload such that a large window is preferred when job arrival rate is high. For typical workload at production supercomputers, we find a window of size 5 makes a good tradeoff between scheduling quality and scheduling overhead.

B. Algorithms

1) Branch and Bound Algorithm: Branch and Bound (B&B) is a general algorithm for finding optimal solutions of discrete and combinatorial optimization problems [24]. In our model, we can control the size of window to limit the number of jobs in the MKP model so that the overhead of the B&B algorithm can be acceptable in terms of time and space complexity. In our experiment, we set window size to 5, which means only the first 5 jobs in the wait queue are considered in our MKP model. The computation time for solving the MKP problem with 5 jobs is affordable for the scheduler. (The scheduler makes scheduling decision periodically with a interval from 10 to 30 seconds.)

In this algorithm, successive levels of the branch-decision tree are constructed by selecting a job and put it into each knapsack in turn. Once the job has been selected for branching, it being put to knapsacks according to the increasing order of knapsacks' indices. After all the knapsacks have been considered, the job is excluded from the current solution. In Figure 3, we give an example of how the optimal solution is found by useing B&B. Each circle represents a state with two arrays indicating the current jobs J and knapsacks K. Here $j_1 = 2$ means job_1 requires 2 nodes and $k_1 = 5$ means the capacity of knapsack k_1 is 5. These two arrays are updated after each decision is made, which generates the searching tree as shown in Figure 3. The circle on the top is the initial state with three jobs and two knapsacks. The B&B algorithm systematically enumerates all candidate solutions by using the upper and lower estimated bounds of quantity being optimized.

Here we use the depth-first search method in B&B. As we can see, two candidate solutions are found in the left bottom of the search tree when at first we put job j_1 to knapsack k_1. Obviously, the solution circled by red line is better where all jobs get allocated and no space in the knapsack is left idle. Also the upper bound of the space left in all knapsacks is set to 0. Based on the upper bound, we can discard other possible decisions such as allocating j_1 to knapsack k_2 or not selecting job j_2. Algorithm 1-2 show the pseudo code of this Branch and Bound algorithm.

Algorithm 1 Branch & Bound

```
E = new (node), this is the dummy start node
H = new (Heap), this is a max-heap for our maximization problem
while true do
    if E is a final leaf then
        E is an optimal solution;
        print out the path from E to the root;
        return;
    end if
    Branch(E);
    if H is empty then
        No solution;
        return;
    end if
    E = delete-top(H);
end while
```

Algorithm 2 Branch

```
Generate all the children of E;
Compute the approximate cost value $\bar{C}_i$ of each child;
Insert each child into the heap H;
```

Branch and Bound algorithm guarantee an optimal solution with an exponential computational complexity of $O(n^m)$. This is feasible due to the small window size. For example, in case of a window size of 5, the algorithm invokes 3,125 solutions which can be solved within a few seconds.

2) Greedy Algorithm: When the window size grows, Branch and Bound algorithm become expensive, we can use the polynomial-time approximate Greedy algorithm. It can obtain a feasible solution by applying the greedy algorithm for classic 0-1 knapsack problem to the first knapsack, then to the second one by using the remaining jobs, and so on. This is obtained by calling m times Algorithm 3. Given the capacity $\bar{C}_i = C_i$ of the current knapsack and the current solution, of value z, stored, for $j = 1, ..., n$, in

$$y_j = \begin{cases} 1 & \text{if job j is currently unassigned;} \\ \text{index of the knapsack it is assigned to.} \end{cases} \tag{10}$$

The solution obtained by calling GREEDY m times can not be optimal. Martello and Toth proposed local exchange techniques that can improve this solution to be optimal [24]. To implement their techniques in our model, we need to do the following things. First, we consider all pairs of jobs put to different knapsacks and try to interchange them if the insertion

Algorithm 3 GREEDY

```
Input: $n, (p_j), (w_j), z, (y_j), i, \bar{C}_i$
Output: $z, (y_j)$
for $j = 1$ to $n$ do
    if $y_j = 0$ and $w_j \le \bar{C}_i$ then
        $y_j = i$;
        $\bar{C}_i = \bar{C}_i - w_j$;
        $z = z + p_j$;
    end if
end for
```

of a new job into the solution is allowed. When all pairs have been considered, we try to exclude in turn each job currently in the solution to replace it with one or more jobs not in the solution so that the total profit is increased. Greedy algorithm has a linear time complexity, i.e., $O(n)$. And the interchange takes $O(n)$ since it only happens when a new job enters the solution. Hence, using GREEDY to find the optimal solution will cost no more than $O(n^2)$ time.

C. An Example

We have the following example to illustrate the difference between our design and the default scheduler using FCFS/EASY backfilling. Here we assume five jobs A, B, C, D, E are submitted and waiting in the queue. The system consists of 20 nodes in total, and the current available nodes are: $\{1, 2, 3, 6, 7, 8, 9, 10, 11, 15, 16, 17, 18, 19, 20\}$. The nodes do not appear in this list (indexes are $4, 5, 12, 13, 14$) are occupied by jobs that are still running. The FCFS/EASY backfilling scheme used by the default scheduler will cut a chunk of six nodes from start of this list for job A, which means node $1, 2, 3, 6, 7, 8$ are assigned to job A. And then sequentially, nodes $9, 10, 11, 15$ will be assigned to B; $16, 17, 18$ to C; 19 to D.

Apparently, this scheme doesn't deliver the best allocation. First, job A and B get a non-contiguous node sets, which means their allocation are fragmented. Node 20 is left idle, and job E has not been satisfied since the available nodes is not enough to satisfy its requirement. Under this default scheduling scheme, the scheduling sequence will always be ⟨A, B, C, D, E⟩, in spite of the current status of node list and each job's size.

Our design first puts these five jobs into the window according to their arrival order, which is A, B, C, D, E. Then it checks the status of the slot list (formed based on system nodes contiguity) and find out how many slots is available. In our example, there are three such slots, $\{1, 2, 3\}$, $\{6, 7, 8, 9, 10, 11\}$, $\{15, 16, 17, 18, 19, 20\}$. Then based on the size of these slots and each job's size, our design will use B&B or Greedy algorithm to make the following scheduling decision. First, it puts job C into the first slot(assign nodes $1, 2, 3$ to C); then put job A into the second slot ($6, 7, 8, 9, 10, 11$ to A); $15, 16, 17, 18$ to B; $19, 20$ to E. Apparently, our design can guarantee that every job gets a compact allocation while maintain high system utilization.

Figure 4 pictorially illustrates this example to highlight the difference between our design and the default scheduler using FCFS policy. As it shows, there are 20 nodes in the node

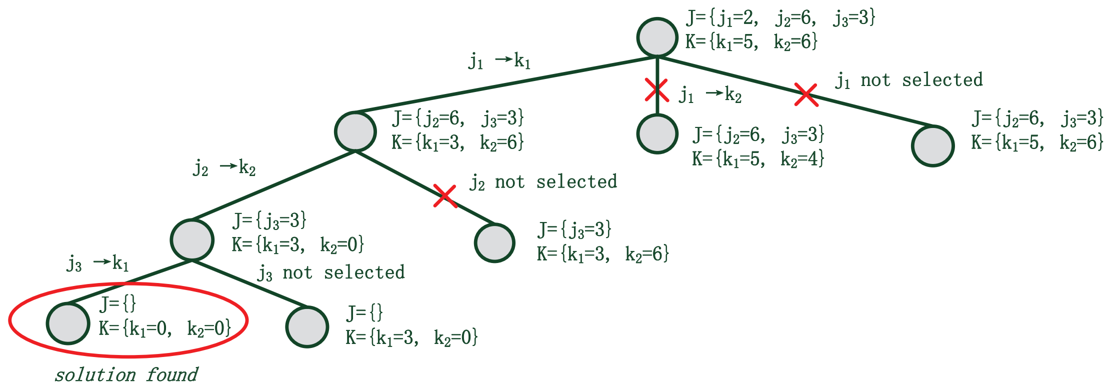

Fig. 3 Decision tree generated for finding the optimal solution by using Branch and Bound Algorithm. There are 2 knapsacks and 3 jobs ($m = 2, n = 3$).

list, the grey ones are been occupied by current running jobs. The subfigure A in 4 is the scheduling result obtained by the default scheduler. It fragments the allocation for job A(with size 6) and job B (with size 4), leaving node No.20 idle and job E (with size 2) unallocated. In subfigure B, our design can guarantee that every job gets a compact allocation.

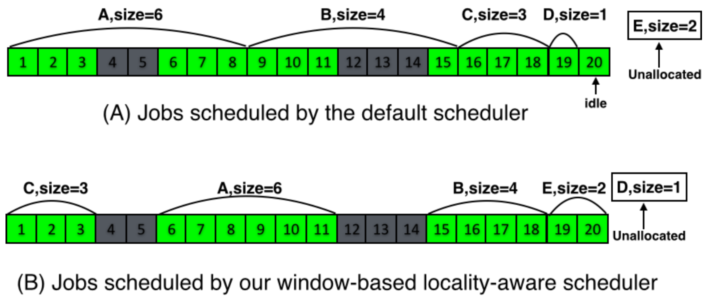

Fig. 4 Scheduling result comparison between the default scheduler and our design. The default scheduler (Subfigure A) makes job prioritizing sequence as $\langle A, B, C, D\rangle$, and the allocation for job A and B are fragmented, node 20 is left idle. Our design (Subfigure B) can make optimization so that every job gets a compact allocation and no node is left idle. The prioritizing sequence obtained by our design is $\langle C, A, B, E\rangle$.

IV. Evaluation Methodology

We conduct a series of experiments using the traces described in Section IV-B to evaluate our design as against the default scheduler using FCFS/EASY backfilling. FCFS/EASY backfilling is the most commonly used scheduling policy on production supercomputers [6][7]. In the rest of the paper, we use *B&B*, *Greedy*, and *Default* to denote our algorithms and the default one. This section describes our evaluation methodology and the experimental results will be presented in the next section.

A. CQSim: Trace-based Scheduling Simulator

Simulation is an integral part of our evaluation of various allocation policies as well as their aggregate effects on system utilization, job's wait time and response time. We have developed a simulator named CQSim to evaluate our design at scale. The simulator is written in Python, and is formed by several modules such as job module, node module, scheduling policy module, etc. Each module is implemented as a class. The design principles are reusability, extensibility, and efficiency. The simulator takes job events from the trace, and an event could be job submission, start, end, and other events. Based on these events, the simulator emulates job submission, allocation, and execution based on specific scheduling and allocation policies. CQsim is open source, and is available to the community [1].

B. Job Traces

In this work, we use two real workload traces collected from production supercomputers to evaluate our design. The objective of using multiple traces is to quantify the performance of our design when dealing jobs and systems with different characteristics. The first trace we used is from a machine named Blue Horizon at the San Diego Supercomputer Center (denoted as SDSC-BLUE in the paper), which contains 4,830 jobs. The second trace we used is from a IBM Blue Gene/P system named Intrepid at Argonne National Laboratory (denoted as ANL-Intrepid in the paper) [3]. This trace contains 2,612 jobs. Figure 5 summarizes job size distribution of these traces. ANL-Intrepid is used to represent capability computing where jobs require a large amount of computing nodes for solving large-scale problems, whereas SDSC-BLUE is used to represent capacity computing where the system is utilized to solve a large number of small-sized problems.

C. Evaluation Metrics

We use three scheduling metrics for evaluation.

- *System Utilization Rate*. This metric denotes the ratio of the node-hours used by jobs to the total elapsed system node-hours. Specifically, let T be the total elapsed time for J jobs, c_i be the completion time for job i and s_i be its the start time, and n_i be the size of job i, then system utilization rate is calculated as

$$\frac{\sum_{0\leq i\leq J}(c_i - s_i)\cdot n_i}{N\cdot T} \tag{11}$$

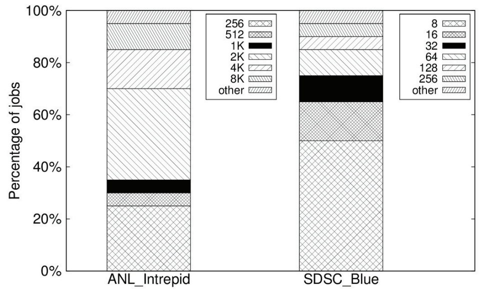

Fig. 5 Job size distribution of ANL-Intrepid and SDSC-BLUE

- *Average Job Wait Time*. For each job, its wait time refers to the time elapsed between the moment it is submitted and the moment it is allocated to run. This metric is calculated as the average across all the jobs submitted to the system. This metric is a user-centric metric, measuring scheduling performance from user's perspective.
- *Average Job Response time*. Response time refers to the amount of time it take when each job is submitted until it ends, which equals to its wait time plus its run time.

V. Experiment Results

We conduct a series of experiments on the traces described in Section IV-B to evaluate our design as against the default scheduler which uses FCFS/EASY backfilling policy.

In our experiments, the scheduler makes scheduling decisions every 30 seconds. To make the scheduler responsive, the window size is set to 5 so that the time cost for B&B algorithm to solve the MKP is affordable (e.g., in seconds). We assume there are β percentage jobs in each trace are sensitive to the contiguity of allocation. Existing studies show that job runtime is influenced by resource allocation, and the variability introduced by different allocations could be as high as 70% [21][15] [17]. In our experiments, we use a parameter α to denote this impact. For a job that is sensitive to the contiguity of resource allocation, we assume its runtime on a contiguous allocation is about α percentage shorter than on a non-contiguous allocation. We conduct a series of sensitivity study to evaluate our design under a variety of configurations. In the following experiments, both β and α are set to $10\%, 20\%, 30\%, 40\%, 50\%$.

A. Evaluation with SDSC-BLUE Trace

The evaluation results for SDSC-BLUE trace are presented in Table I, II, III. Table I shows the system utilization improvement obtained by our design using B&B and Greedy as against the default scheduler using FCFS/EASY backfilling. In general, our design can outperform the default scheduler by about 1% to 3%. As the impact parameter α and job percentage β increase, this improvement grows slowly. Apparently, the system's throughput is not sensitive to the growth of jobs' running time since those jobs only required a very small portion of system nodes.

TABLE I System utilization improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	0.98% 1.15%	1.37% 1.42%	1.69% 1.52%	1.66% 1.68%	1.92% 1.84%
20%	1.28% 1.11%	1.09% 1.12%	1.46% 1.54%	1.50% 1.63%	1.77% 1.69%
30%	2.23% 2.18%	2.23% 2.21%	2.35% 2.35%	2.48% 2.43%	2.54% 2.67%
40%	2.36% 2.28%	2.49% 2.49%	2.64% 2.47%	2.84% 2.84%	3.03% 2.92%
50%	3.10% 3.07%	3.14% 3.20%	3.31% 3.48%	3.72% 3.68%	3.83% 3.83%

Table II shows job average wait time improvement obtained by our design using B&B and Greedy as against the default scheduler. As the impact parameter α and job percentage β increase, the improvement can be as much as 27%. This result indicates when large portion of jobs suffer from adverse impact introduced by inappropriate node allocation, our design using B&B and Greedy can greatly outperform the default scheduler.

TABLE II Average job wait time improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	11.27% 11.52%	11.71% 11.42%	12.95% 12.62%	14.57% 14.46%	17.17% 18.31%
20%	11.36% 12.07%	12.54% 12.44%	13.99% 14.55%	16.50% 16.22%	18.20% 19.05%
30%	12.98% 12.55%	13.87% 13.81%	14.04% 12.98%	16.16% 16.66%	19.29% 20.31%
40%	14.61% 15.30%	15.86% 15.57%	16.22% 16.48%	19.48% 19.31%	22.36% 21.70%
50%	19.35% 18.52%	21.98% 21.70%	23.13% 23.84%	25.29% 25.05%	27.33% 26.86%

Table III shows the improvement of average job response time obtained by our design using B&B and Greedy as against the default scheduler. This metric has the same trend as *average job wait time* and reaches even higher value. This is because job's run time is usually longer than its wait time in SDSC-BLUE trace and most jobs have their runtime dominate the total response time. Thus, the impact parameter α has much greater influence to jobs' response time than to wait time.

TABLE III Average job response time improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	10.47% 9.27%	10.14% 10.29%	10.28% 10.74%	11.11% 11.20%	12.62% 12.32%
20%	11.12% 11.07%	12.29% 12.23%	12.93% 13.10%	13.33% 13.37%	14.93% 14.78%
30%	12.63% 12.56%	13.73% 13.68%	15.01% 15.25%	16.20% 16.12%	17.79% 17.32%
40%	16.17% 16.03%	16.81% 16.98%	17.09% 17.26%	17.90% 17.93%	18.90% 18.76%
50%	18.81% 18.17%	22.11% 22.00%	23.39% 23.45%	25.56% 24.98%	28.83% 28.41%

B. Evaluation with ANL-Intrepid Trace

The evaluation results for ANL-Intrepid trace are presented in Table IV, V and VI. Table IV shows that the system utilization improvement obtained by our design from ANL-Intrepid trace can be as much as 4.8%, which is slightly higher than that from SDSC-BLUE. This is because two traces have different job size distribution, as shown in IV-B. The variation of job size in ANL-Intrepid trace is greater than SDSC-BLUE. The smallest jobs in ANL-Intrepid require 256-node, while the biggest jobs require 8K nodes. When a big job released from the system, it vacates a very big slot with great potential for our design to make optimization. Hence, the system utilization improvement obtained from ANL-Intrepid trace is higher than from SDSC-BLUE trace.

TABLE IV System utilization improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	0.42% 0.54%	0.53% 0.56%	0.61% 0.63%	0.89% 0.84%	1.04% 1.12%
20%	1.02% 1.18%	1.28% 1.24%	1.47% 1.32%	1.78% 1.83%	2.20% 2.28%
30%	3.11% 3.20%	3.21% 3.20%	3.35% 3.32%	3.35% 3.30%	3.34% 3.35%
40%	3.00% 3.02%	3.14% 3.10%	3.23% 3.23%	3.23% 3.25%	3.24% 3.31%
50%	4.12% 4.12%	4.35% 3.35%	4.38% 4.23%	4.42% 4.57%	4.84% 4.69%

Both average job wait time and response time improvement obtained by our design from ANL-Intrepid trace is not as prominent as from SDSC-BLUE. As shown in Figure 5, more than 50% jobs in the ANL-Intrepid trace are of size 1K, 2K, 4K, which means job size variation within these jobs are relatively small. And these jobs have much more longer running time than those small jobs in SDSC-BLUE, which makes their wait time not as sensitive to the impact parameter α as jobs in SDSC-BLUE trace. The average wait time and response time got by our design from ANL-Intrepid trace is about 10%, shown in Table V and VI.

TABLE V Average job wait time improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	6.32% 6.10%	7.12% 7.18%	7.39% 7.60%	7.54% 7.51%	8.84% 7.92%
20%	6.20% 6.16%	7.87% 7.37%	8.87% 8.59%	8.10% 8.60%	9.12% 8.94%
30%	7.04% 6.97%	7.89% 7.60%	8.33% 8.46%	9.58% 9.73%	10.69% 10.77%
40%	8.27% 8.34%	8.81% 8.86%	9.62% 9.60%	10.06% 9.98%	10.89% 10.76%
50%	8.45% 8.62%	9.09% 9.15%	10.13% 10.16%	10.65% 10.62%	11.68% 11.54%

TABLE VI Average job response time improvement obtained by our design using B&B and Greedy as against the Default scheduler. In each cell, the number on top is the improvement achieved by using B&B, the bottom number is the improvement achieved by using Greedy.

	Impact Parameter α				
Job Percentage β	10%	20%	30%	40%	50%
10%	3.81% 4.00%	4.09% 4.26%	4.87% 4.76%	5.05% 4.93%	5.76% 5.81%
20%	5.16% 4.03%	5.53% 4.46%	5.89% 5.76%	6.76% 6.76%	8.91% 8.80%
30%	5.03% 5.16%	5.52% 5.81%	6.76% 6.86%	7.90% 7.67%	8.93% 9.00%
40%	6.08% 5.70%	7.52% 7.35%	7.73% 7.58%	8.89% 8.74%	9.54% 9.26%
50%	6.16% 6.05%	7.09% 7.10%	9.12% 9.23%	9.90% 9.85%	10.85% 10.68%

C. Result Summary

In summary, our trace-based experiments have shown the following:

- Our window-based locality-aware scheduling design can guarantee compact job allocation while maintaining high system utilization. The experimental results also demonstrate that our design is capable of reducing average job wait time and job response time.
- Our design can deliver up to 27% reduction on average job wait time and response time, and 4% improvement on system utilization. The amount of improvement

varies depending on workload features such as job size and job running time.

- Both B&B and GREEDY algorithms can deliver comparable performance in our case studies. Considering the computational overhead, we recommend the use of GREEDY due to its low computational complexity.

VI. Related Work

The most commonly used scheduling policies are FCFS combined with EASY backfilling [5]. Under this policy, the scheduler picks a job from the head of the wait queue and dispatches it to the available system resources. Many studies seek to improve the performance of this classic scheduling paradigm. Tang et al. made refinement about user's estimated job runtime in order to make the backfilling more efficient [6] [8]. They also designed a walltime-aware job allocation strategy, which adjacently packs jobs that finish around the same time, in order to minimize resource fragmentation caused by job length discrepancy [7]. And there are some other variation of FCFS/EASY backfilling proposed to optimize system performance in terms of power consumption and energy cost [9][10]. However, none of them ever take allocation locality into consideration when making scheduling decisions.

There are several studies focusing on allocation algorithms to minimize system fragmentation. Lo et al. presented a non-contiguous allocation scheme named Multiple Buddy Strategy (MBS) [14]. MBS preserves locality by allocating each job a set of "blocks" to reduce interference between jobs, with the advantage of eliminating both internal and external fragmentation. Each "block" consists of 2^n nodes that adjacent to each other (n with different value depends on the block size). However, the distance between "blocks" could be too long to make the communication between processes within the same application less efficient. MBS also needs to partition the system into fixed number of "blocks" in advance, which is time consuming and low-efficient for big scale systems.

Leung et al. presented allocation strategies based on space filling curves and one dimensional packing [22]. They implemented these strategies using 2-dimensional Hilbert curves and had an integer program for general networks. Their preliminary experimental results show that processor locality can be preserved in massively parallel supercomputers using one-dimensional allocation strategies based on space filling curve. However, space filling curve has the limitation that it can only be applied to system with the scale of 2^n nodes in each dimension.

Albing et al. conducted study about the allocation strategies that the Cray Application Level Placement Scheduler (ALPS) used [23]. The job allocation in Cray Linux Environment (CLE) operating system is managed by ALPS, which works from a list of available nodes and assigns those nodes in sequence from this list to jobs. However, ALPS does not make changes or calculations when making allocation decisions. It just simply works off the ordered list, however that is ordered. They claimed that the ordered list can be obtained by using either Hilbert curve or simply sorting the nodes based on their spacial coordinates in the system.

There are other studies focusing on allocation algorithms to improve the performance of user jobs. Pascual et al. proposed an allocation strategy that aiming to assign a contiguous allocation to each job, in order to improve communication performance [12]. However, this strategy results in severe scheduling inefficiency due to increased system fragmentation. They reduced this adverse effect by using a relaxed version called quasi-contiguous allocation strategy.

Another related work is online bin packing. In the bin packing strategy, the objective is to pack a set of items with given sizes into bins. Each bin has a fixed capacity and cannot be assigned to items whose total size exceeds this capacity. The goal is to minimize the number of bins used. The off-line version is NP-hard [18] and bin packing was one of the first problems to be studied in terms of both online and offline approximability [19][20].

This work has two major difference as compared to the above literatures. First, rather than one-by-one job scheduling as most existing schedulers do [23][14][12], our design takes a "window" of jobs (i.e.,multiple jobs) into consideration for job prioritizing and resource allocation. Second, our resource management module takes a contiguous set of nodes as a *slot* and maintains a list of such slots. The slot list is updated dynamically when job being allocated/deallocated in the system. The allocation of the jobs in the window onto the slot list is conducted in such a way as to maximize system utilization. This is different from the existing job allocation schemes that work off a ordered list, which loses the spacial information of the torus-connected system.

VII. Conclusions

In this paper, we have presented a window-based locality-aware job scheduling design for torus-connected system. Our goal is to balance job performance with system performance. Our design has three novel features. First, rather than one-by-one job scheduling, our design takes a "window" of jobs (i.e.,multiple jobs) into consideration for job prioritizing and resource allocation. Second, our design maintains a list of slots to preserve node contiguity information for resource allocation. Finally, we formulate a 0-1 Multiple Knapsack problem to describe our scheduling decision making and present two algorithms to solve the problem. Preliminary results based on trace-based simulation demonstrate our design can reduce average job wait time by up to 28% and average job response time by 30%, with a slight improvement on overall system utilization.

Our future work includes experiments with various workload traces collected from production systems as well as different network topologies in addition to torus (e.g., fat-tree, dragonfly). We intend to conduct quantified analysis about different allocation schemes in terms of system performance and user job's performance so that the best allocation schemes can be chosen for HPC systems with different network topologies.

Acknowledgment

We appreciate the valuable comments and suggestions from the anonymous reviewers. The work at Illinois Institute of Technology is supported in part by US National Science Foundation grant CNS-1320125. The work at Argonne is supported in part by the U.S. Department of Energy (DOE), Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

[1] Cqsim: An event-driven simulator. http://bluesky.cs.iit.edu/cqsim

[2] IBM redbooks publication, IBM system Blue Gene solution: Blue Gene/Q system administration.

[3] Parallel Workload Archive. http://www.cs.huji.ac.il/labs/parallel/workload

[4] Managing System Software for Cray XE and Cray XT Systems. Cray Document. 2012.

[5] D. Feitelson and A. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *International Parallel and Distributed Processing Symposium*, 1998.

[6] W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In *2010 IEEE International Symposium on Parallel Distributed Processing*, 2010.

[7] W. Tang, Z. Lan, N. Desai, D. Buettner, and Y. Yu. Reducing fragmentation on torus-connected supercomputers. In *2011 IEEE International Symposium on Parallel Distributed Processing Symposium*, 2011.

[8] W. Tang, N. Desai, D. Buettner, and Z. Lan. Job Scheduling With Adjusted Runtime Estimates on Production Supercomputers. Journal of Parallel and Distributed Computing (JPDC), 73(7):926-938, 2013.

[9] Z. Zhou, Z. Lan, W. Tang, and N. Desai. Reducing energy costs for IBM Blue Gene/P via Power-Aware job scheduling. In *17th Workshop on Job Scheduling Strategies for Parallel Processing*, 2013.

[10] X. Yang, Z. Zhou, S. Wallace Z. Lan, W. Tang, S. Coghlan and Mike. Papka. Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems. In *2013 ACM/IEEEE Supercomputing*, 2013.

[11] P. Krueger, T. Lai, and V.A. Dixti-Radiya Job Scheduling Is More Important than Processor Allocation for Hypercube Computers IEEE Trans. Parallel and Distributed Systems, vol. 5, no. 5, page 488–497, May 1994.

[12] JA. Pascual, J. Navaridas and J. Miguel-Alonso. Effects of Topology-Aware Allocation Policies on Scheduling Performance. 14th Workshop on Job Scheduling Strategies for Parallel Processing. May 29, 2009.Rome, Italy.

[13] D.G. Feitelson, L. Rudolph and Schwiegelshohn, Parallel job scheduling status report. In *Job Scheduling Strategies for Parallel Processing, Springer Verlag (2005)* . pages 1–116

[14] V. Lo, K. Windisch, W. Liu, and Nitzberg. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. IEEE Transactions on Parallel and Distributed Systems 8 (1997). pages 712–726

[15] A. Bhatele, L.V. Kale Application-specific topology-aware mapping for three dimensional topologies. In *Proceedings of Workshop on Large-Scale Parallel Processing (held as part of IPDPS' 08)* 2008

[16] Y. Aridor, T. Domany, O. Goldshmidt and J.E. Moreira. Resource allocation and utilization in the Blue Gene/L Supercomputer. IBM Journal of Research and Development 49 (2-3) (2005) pages 425–436

[17] Ansaloni, R. The Cray XT4 Programming Environment. http://www.csc.fi/ english/csc/courses/programming/

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

[19] D. S. Johnson. Near-optimal Bin Packing Algorithms. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1973.

[20] D. S. Johnson. Fast Algorithms for bin Packing. J. Comput. Syst. Sci., 8:272–314, 1974.

[21] Bhatele, Abhinav and Mohror, Kathryn and Langer, Steven H. and Isaacs, Katherine E. There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs In Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, 2013

[22] Leung, V.J.; Arkin, E.M.; Bender, M.A.; Bunde, D.; Johnston, J.; Alok Lal; Mitchell, J.S.B.; Phillips, C.; Seiden, S.S., Processor allocation on Cplant: achieving general processor locality using one-dimensional allocation strategies In Proceedings of 2002 IEEE International Conference on Cluster Computing, 2002. pages 296–304, 2002

[23] Carl Albing and Mark Baker ALPS,Topology, and Performance: A Comparison of Linear Orderings for Application Placement in a 3D torus. Presented at the CUG 2010, Edinburgh, Scotland, UK, 2010.

[24] S. Martello, P. Toth. Heuristic algorithms for the multiple knapsack problem. Computing 27, 93112. 1981.