International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Research Paper

Overhead of using spare nodes

The International Journal of High Performance Computing Applications
2020, Vol. 34(2) 208–226
© The Author(s) 2020
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342020901885
journals.sagepub.com/home/hpc
SAGE

Atsushi Hori[1], Kazumi Yoshinaga[2], Thomas Herault[3], Aurélien Bouteiller[3], George Bosilca[3] and Yutaka Ishikawa[1]

Abstract
With the increasing fault rate on high-end supercomputers, the topic of fault tolerance has been gathering attention. To cope with this situation, various fault-tolerance techniques are under investigation; these include user-level, algorithm-based fault-tolerance techniques and parallel execution environments that enable jobs to continue following node failure. Even with these techniques, some programs with static load balancing, such as stencil computation, may underperform after a failure recovery. Even when spare nodes are present, they are not always substituted for failed nodes in an effective way. This article considers the questions of how spare nodes should be allocated, how to substitute them for faulty nodes, and how much the communication performance is affected by such a substitution. The third question stems from the modification of the rank mapping by node substitutions, which can incur additional message collisions. In a stencil computation, rank mapping is done in a straightforward way on a Cartesian network without incurring any message collisions. However, once a substitution has occurred, the optimal node-rank mapping may be destroyed. Therefore, these questions must be answered in a way that minimizes the degradation of communication performance. In this article, several spare node allocation and failed node substitution methods will be proposed, analyzed, and compared in terms of communication performance following the substitution. The proposed substitution methods are named *sliding methods*. The sliding methods are analyzed by using our developed simulation program and evaluated by using the K computer, Blue Gene/Q (BG/Q), and TSUBAME 2.5. It will be shown that when failures occur, the stencil communication performance on the K and BG/Q can be slowed around 10 times depending on the number of node failures. The barrier performance on the K can be cut in half. On BG/Q, barrier performance can be slowed by a factor of 10. Further, it will also be shown that almost no such communication performance degradation can be seen on TSUBAME 2.5. This is because TSUBAME 2.5 has an Infiniband network connected with a FatTree topology, while the K computer and BG/Q have dedicated Cartesian networks. Thus, the communication performance degradation depends on network characteristics.

Keywords
Fault tolerance, fault mitigation, spare node, communication performance, sliding method

1. Introduction

With the fault rate increasing on high-end supercomputers, the topic of fault tolerance has been gathering attention (Cappello et al., 2014), and jobs are being aborted due to system errors (Di Martino et al., 2014). To cope with this situation, various fault-tolerance techniques have been investigated. Checkpoint and restart is a well-known technique for parallel jobs, and enabling jobs to continue execution from a previously defined checkpoint (there are many studies and systems of checkpoint and restart, but the most notable one is Checkpointing Libraries for the Intel Paragon (CLIP); Chen et al., 1997).

With the increase in size of parallel applications, the total amount of Input/Output (I/O) needed for checkpoint/restart begun to be problematic. A lot of research is currently undertaken on techniques to reduce the checkpoint amount in order to alleviate the I/O issue (e.g. Sato et al., 2012). On the other hand, user-level checkpoints, where each program implements its own checkpoint/restart strategy, have been attracting attention as a possible alternative. Since the user knows which data should be saved and which data can be lost, the amount of checkpoint data can be drastically reduced, and thus the I/O time can also be

[1] RIKEN Center for Computational Science, Kobe, Hyogo, Japan
[2] Meguro-ku, Tokyo, Japan
[3] Innovative Computing Laboratory, The University of Tennessee, Knoxville, TN, USA

Corresponding author:
Atsushi Hori, RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan.
Email: ahori@riken.jp

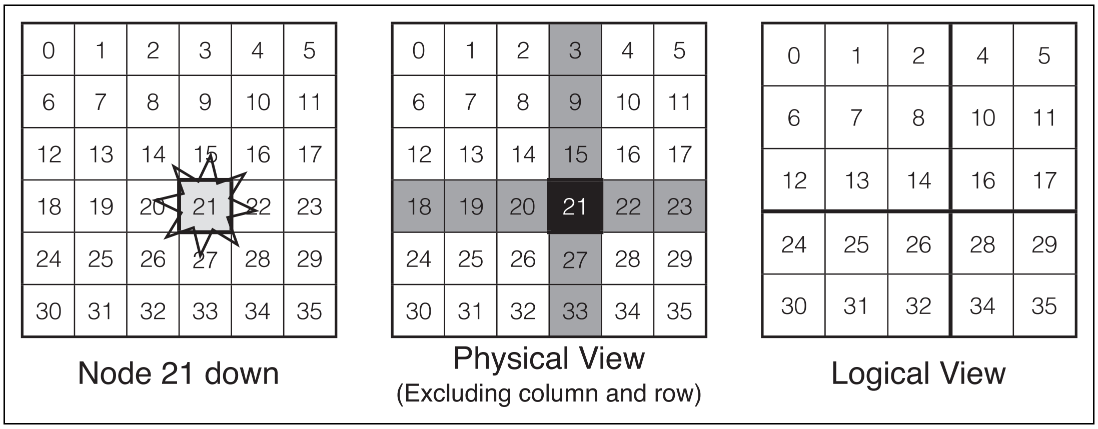

Figure 1. Example of node failure and recovery.

greatly reduced, at the cost of only some additional programming by the user.

Davies et al. (2011) presented a method that allows a user program to be fault-tolerant without using checkpointing (Chen and Dongarra, 2008). In this technique, the parity to recover the lost data can be embedded into an lower-upper (LU) decomposition algorithm, and the user program can recover from failure without checkpointing. Having the opportunity to address the failure at the algorithm level opens interesting perspective and new research topics. With support from the programming paradigm and the execution environment, users could write applications which can handle faults in the most optimal way.

The Message Passing Interface (MPI) is the most widely used communication library, and its specifications are well defined (Message Passing Interface Forum, 2012). Unfortunately, in the current MPI standard, a fatal error handler is raised upon process failure, preventing any user-level fault handling to be implemented at this time. To define the behavior of MPI when a fault occurs, User-Level Failure Mitigation (ULFM) has been proposed and a prototype is being developed, capable of handling both process and node failures (Bland et al., 2013). ULFM provides the application program interface (API) so that the modifications to the existing MPI specifications are minimized. Even with ULFM, user-level fault handling is not straightforward, and various frameworks have been proposed to simplify it. Falanx is a fault-tolerant framework for master-worker programming (Takefusa et al., 2014). Local Failure Local Recovery (LFLR) is another fault-tolerant framework (Teranishi and Heroux, 2014), and it covers a wider range of programming models than are supported by Falanx. Both Falanx and LFLR are implemented by using ULFM. Global View Resilience (GVR) is another user-level fault mitigation system; it is based on partitioned global address space programming model (Chien et al., 2015; Fujita et al., 2014).

We believe that the user-level fault-handling code must be as simple as possible. It is important to avoid situations in which the code for handling the first node failure is different from the code for handling subsequent failures, because it is very hard to produce this type of situation when testing a program. This type of complexity must be hidden within the system software.

Figure 1 shows an example of a node failure in a 2D network consisting of 36 nodes. Here, it is assumed that a job is running on this machine, and the job is written with a fail-stop-free runtime system, such as ULFM. When node 21 goes down (left panel in Figure 1), the job running on those 36 nodes can take one of the following actions:

- abort the job and resubmit it (from a previous checkpoint, if possible), or
- allow the remaining 35 nodes to continue to execute the job.

In the first strategy, user-level fault handling is not required. In the second strategy, the task allocated to the failed node must be equally shared by the remaining 35 nodes, otherwise, a load imbalance occurs. If the job can balance a dynamic load, which is a capability of master-worker models and particle-in-cell simulations, then the load can be rebalanced by the application itself, without the need to extensively modify the code. However, if the job is a stencil application, which, in most cases, does not have dynamic load balancing capability, then fault handling is more difficult. In most stencil applications, both the communication pattern and the load balancing are static. To preserve the communication pattern, one possibility for handling a node failure is to exclude the row and column that include the failed node (middle panel in Figure 1); this preserves the stencil communication pattern. However, the task allocated to the failed node must be shared equally by the remaining nodes (right panel in Figure 1). This load-leveling requires additional code for handling the fault, and this must be avoided if possible.

If a system software reserves a set of spare nodes in advance, and the failed node is replaced by a spare node, then the user-level handling of node failure is simplified, because the number of nodes involved in the computation remains the same. LFLR assumes the use of spare nodes, and although the detailed recovery process is hidden from users, GVR may utilize spare nodes. However, to the best of our knowledge, there has been no discussion of the best way to reserve spare nodes or of how to use them to replace failed nodes. As an evaluation index, we chose communication performance, because the use of spare nodes may introduce extra message collisions.

The scenario we assumed to recover from a node failure is; (1) user program detects a node failure, (2) select a spare node from a spare node set to substitute the failed node, (3) recover the process(es) on the spare node possibly from a checkpoint, (4) rearrange the physical process-to-node mapping to minimize the communication performance degradation and MPI communicator(s) to minimize the change of user program, (5) if the new process-to-node mapping requires migration of some processes, do the migration, and (6) finally, the user program resumes its execution.

This article presents the results of our investigations into these issues. As a first step to address these issues, we propose several methods for using spare nodes to replace faulty ones in addition to the spare node allocations. The proposed methods are discussed and compared from the viewpoint of communication performance degradation. The contributions of this article are as follows:

- spare node allocation methods are proposed;
- failure node substitution methods are proposed;
- focusing on stencil communication and some collective performance, the behavior and characteristics of proposed spare node allocation and substitution methods are revealed by simulations; and
- evaluations results on the two supercomputers having a Cartesian network topology and one supercomputer having a FatTree network topology are shown to how a network topology affects the communication performance degradation.

2. Using spare nodes

For the remainder of this article, we will assume that the networks being considered have a multidimensional Cartesian (mesh and/or torus) topology, otherwise noticed. We make this assumption because four of the top five machines have networks with this topology (as listed on the TOP500 Supercomputer Site; Strohmaier et al., 2015); see Table 1.

From the programmers' point of view, it is not complicated to have spare nodes held ready or to have them substituted in for faulty nodes. With MPI, the modification is as follows: (1) a new MPI communicator is created at the location from which the faulty node is extracted (in ULFM, the command `MPI_Comm_shrink` will do this), and a selected spare node replaces the faulty node; (2) the spare node is set up to take over the functions of the failed node. The remaining parts of the program can remain as they were. This means that the logical topology provided by the new MPI communicator can remain the same as it was before the failure; however, the actual physical topology is altered. New message collisions that would not have

Table 1. Network topologies in the TOP500 list.

Rank	Name	# Cores	Topology
1	Tianhe-2	3120K	FatTree
2	Titan (Cray XK7)	561K	3D torus
3	Sequoia (BG/Q)	1573K	5D torus/mesh
4	The K computer	705K	6D torus/mesh
5	Mira (BG/Q)	786K	5D torus/mesh
11	JUQUEEN (BG/Q)	459K	5D torus/mesh
25	TSUBAME 2.5	76K (+GPU)	FatTree

Source: Strohmaier et al., 2015.
BG/Q: Blue Gene/Q; GPU: graphics processing unit.

happened under the failure-free physical topology will happen under the recovered topology (Figure 4).

Therefore, replacing faulty nodes with spare nodes must be done carefully in order to minimize the communication performance degradation. There are many other aspects that should be considered, such as system utilization, job turnaround time, ease of user programming, and the framework that needs to be developed. Unfortunately, almost no research has been done on this topic, so in this article, we will focus primarily on the communication performance.

Throughout this article, we will be concerned only with the node failure. Network failures can also occur, but we will assume that this recovery is the responsibility of the network itself (Domke et al., 2014; see also Section 5). The Tofu network, which is used by the K computer, uses redundant links to detour around failed nodes (Ajima et al., 2009; Sumimoto, 2012). We will assume that a job can survive even with the failure of one or more nodes when it is operating in a parallel computing environment that provides a user-level fault mitigation mechanism, such as ULFM, and any processes running on the failed node can be recovered from a checkpoint or by using parity with viable processes. Finally, we will assume that the processes running on a node can be migrated to any other node.

In the next subsection, we will discuss the allocation of spare nodes, and the possibility of this degrading the communication performance will be shown. Then, three methods for substituting a spare node for a faulty node will be proposed and compared.

2.1. Spare node allocation

In this section for simplicity, we will consider mostly 2D networks with static XY routing (Zhang et al., 2009). Figure 2 shows three different ways of allocating spare nodes. Each small square represents a node. In the left panel, the right-hand column is reserved for spare nodes; this pattern is denoted as 2D(1,1). In the middle panel, two sides (the right-hand column and the bottom row) are reserved for spare nodes, denoted 2D(2,1). In the right-hand panel, two two-node thick sides (the two right-hand columns and two bottom rows) are reserved, denoted 2D(2,2). In this notation, “2D” means that the allocation applies to the 2D plane, the first number in the brackets is

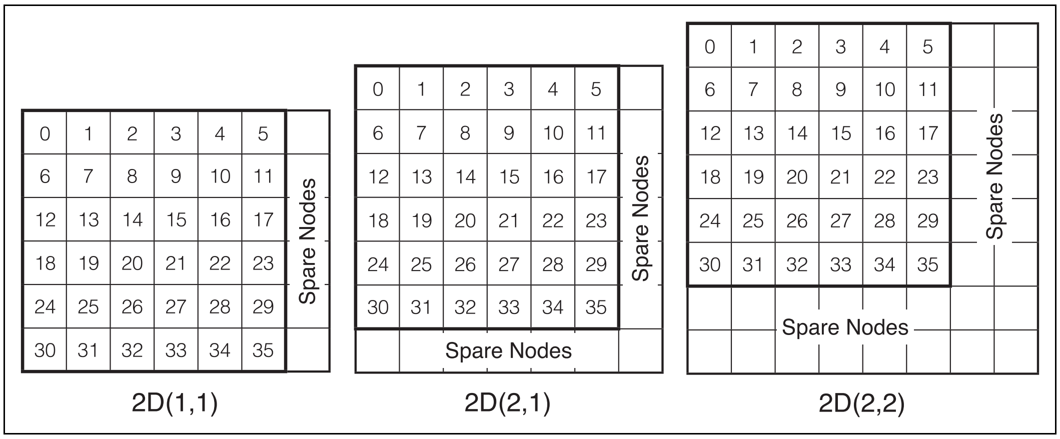

Figure 2. Patterns for allocation of spare nodes.

the number of sides in which spare nodes are reserved, and the second number is the thickness, number of columns or rows, of a side of spare nodes reserved. On a 3D network, a job is generally allocated on a cube. Here, 3D(2,*) means two 2D side planes out of three dimensions of the cube are allocated as spare nodes. If the allocated cube is regular, having the same size in all dimensions, which side to be allocated does not matter. Otherwise, the number of spare nodes in the two side planes matters.

Spare nodes are allocated at the side(s) of a 2D grid, as shown in Figure 2; thus, a stencil application with non-periodic boundaries will not have any overhead. This will not be the case for stencil applications that have periodic boundaries or for networks that have torus topology. However, the hop count is only increased by one, so the increase in run time will be very small (100 ns per hop on the K computer).

The percentage of the nodes that are reserved as spare nodes in the 2D(2,2) case is as follows

$$R_{2D(2,2)} = 1 - \left(N^{1/2} - 2\right)^2 \Big/ N$$

where N is the number of nodes. In the more general $q\mathrm{D}(r,s)$ case, the percentage of spare nodes can be expressed as follows

$$R_{q\mathrm{D}(r,s)} = 1 - \frac{(N^{1/q} - s)^r \times (N^{1/q})^{q-r}}{N}$$

Here, $r \leq q$ and AH$s < N^{1/q}$. Note that this expression is not precise, because the number of nodes is an integer, and the flooring effect is ignored. However, this information can be useful for determining how the spare node percentage relates to the total number of nodes used for a job.

Figure 3 shows the percentages of spare nodes to the whole nodes allocated for a job over the various patterns of spare node allocation. As shown in this figure, the more dimensions the network has, the higher the percentage of spare nodes. The percentage is almost proportional to the number of sides allocated to the spare nodes. Most notably, the larger the job size, the lower the percentage. We will discuss this point in Section 6.

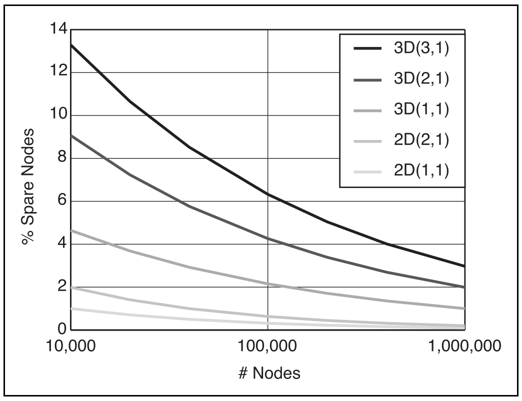

Figure 3. Percentage of spare nodes in a job.

It is possible to allocate spare nodes on four sides of a 2D grid, but on a torus network, this is almost equal to the 2D(2,2) case. In our investigation, we could not find any significant difference between 2D(4,1) and 2D(2,2), and so in this discussion, we will not further consider cases in which $r > q$. The thickness, s, does not affect the nature of the spare node substitution method described in the next section, so we will investigate only cases of single-node thickness.

Having spare nodes can decrease the system utilization ratio. However, this does not always happen. On the K computer, the size of each dimension of a job must be in a *Tofu unit*, which has 12 nodes. When a user submits an 11 × 11 × 11 3D job, for example, it may be scheduled to have 12 × 12 × 12 nodes. This results in 3D(2,1) spare nodes. The same situation can be seen with the other machines that have a Cartesian topology network and are listed in Table 1. On Blue Gene/Q (BG/Q) machines, the number of nodes for a job must be a power of 2 (IBM, 2013b). On a Cray XK/7, jobs are allocated to 428 blocks (Peña et al., 2013). Thus, the gap between the number of nodes required by a job and the number of nodes actually allocated can be allocated as spare nodes, without requiring additional nodes.

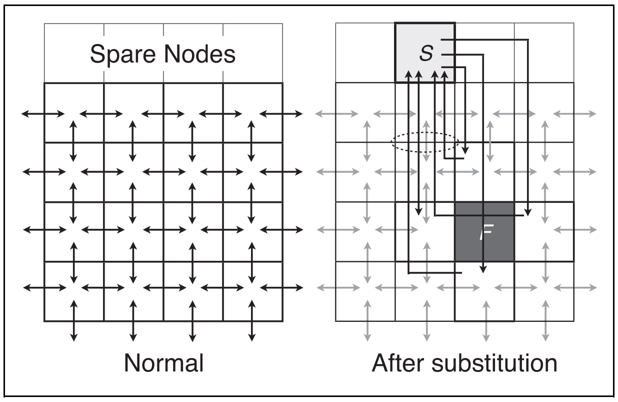

Figure 4. Message collisions.

2.2. Substitution of a spare node for a faulty node

Communication performance degradation can be observed when a spare node that replaces a faulty node can be located far from the original node. Figure 4 shows the 5P-stencil communication pattern (left). In 5P-stencil communication on a Cartesian topology, no messages collide, because nodes communicate only with their neighbors. Here, static XY routing is assumed. In the right-hand panel of Figure 4, when a faulty node (denoted as "*F*") is replaced by a spare node (denoted as "*S*"), the regularity of the stencil communication pattern is lost. As shown in this figure, there are five message routes crossing through the circled link, this means that up to five messages can collide.

We propose three methods for substituting nodes, and these are shown in Figure 5. We call these methods the 0D, 1D, and 2D sliding methods. With higher-dimension networks, those proposed methods can be augmented in a natural way, but for simplicity, we will explain them on a 2D network. We will use a 5P-stencil communication pattern, in which messages from each node are sent up, down, left, and right. In the 9P-stencil communication pattern, there are an extra four directions, since messages can be sent along the diagonals. However, in most cases, the length of those diagonal messages is much shorter than those in a 5P-stencil pattern, and so the effect on the communication performance is expected to be small.

2.2.1. 0D sliding. The 0D sliding method is the simplest. The faulty node is simply replaced by a spare node (as was shown in Figure 5). There is a big drawback to this method, however, when a node failure happens far from a spare node: the hop distance from the failed node to the spare node can be very large. This increases the possibility of message collisions and results in a higher communication latency due to the large number of hops. To minimize this, the failed node should be replaced with the spare node to which the Manhattan distance is the shortest.

Figure 6 shows examples of the results of replacing multiple faulty nodes when using the 0D sliding method with the 2D(1,1) allocation. In this figure, each rectangle represents a node, assuming node space can be represented in a 2D way. On the left-hand panel, nodes 1 through 5 have failed and have been replaced by spare nodes 1′ through 5′, respectively. The spare nodes were chosen so as to minimize the number of hop counts between each faulty node and its corresponding spare node. With nonperiodic 5P-stencil communication in the XY routing algorithm, the messages from all of the spare nodes to the nodes (A through F) adjacent to the failed nodes are routed through node 1′ (because of the X direction routing of the XY routing algorithm). Thus, there are 11 messages in the network links between 1′ and A (these are shown in the white boxes): these 10 plus the normal stencil communication message between the nodes. This is the worst-case scenario for the 0D sliding method, and the number of faulty nodes is less than or equal to six.

The right-hand panel of Figure 6 shows a case for which the network topology is a 2D mesh, spare nodes are reserved in the 2D(1,1) pattern, and the faults happen within a row or column that is close to the side of the network. Failed node 1 is replaced by spare node 1′, and so on. In this case, the failures happen close to the side of the network, and it is not possible to replace the spare nodes as in the left-hand panel of Figure 6. In nonperiodic 5P-stencil communication, all messages from spare nodes 4′, 5′, 6′, and 7′ to the neighbor nodes A to V go through the link between 3′ and 4′. There are 16 messages, since each node sends 4 messages, one to each of its neighbor nodes. This situation can happen when the number of faults is greater than or equal to seven. Below, we state the relation between the maximum number of possible message collisions ($C_{\max}$) and the number of node failures (F_n). Note that when $C_{\max}$ is equal to one, then there is only one message on each network link, and there are no collisions

$$C_{\max} = \begin{cases} 2 \times F_n + 1 & F_n \le 6 \text{ or torus topology} \\ 4 \times (F_n - 3) & F_n \ge 7 \text{ and mesh topology} \end{cases}$$

This worst-case scenario can be relaxed by having spare nodes allocated in the 2D(2,1) pattern. If the failures happen in the same row or column, then the spare nodes must be chosen from alternating sides.

2.2.2. 1D sliding. As described in the previous subsection, in the 0D sliding method, even if the closest spare node is chosen, the distance from the failed node is unlikely to be small. The 1D sliding method can avoid this situation, and it is shown in Figure 7. When node 21 fails, instead of replacing it with a spare node, the nodes of the column (or row) that include the failed node shift toward a spare node, as shown in the upper left-hand panel of the figure. In this way, the hop count in the 5P-stencil communication pattern is increased by only one. This is much smaller than occurs with the 0D sliding method.

In terms of hop counts, the 1D sliding method is superior to the 0D sliding method; however, the recoverable

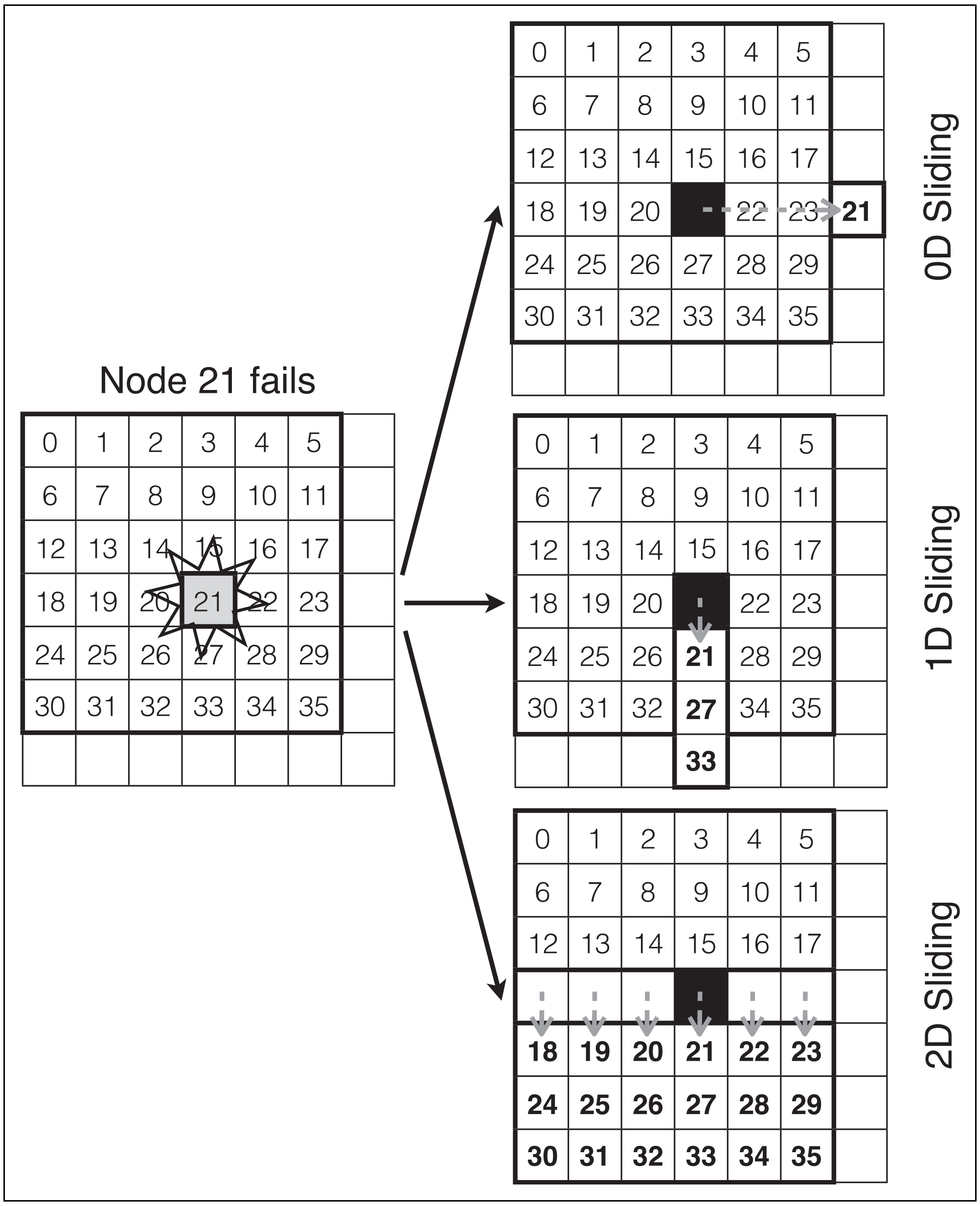

Figure 5. Substitution methods for faulty nodes.

number of faulty nodes is limited in some cases. Let us consider a case in which a second node (16) fails (again using the 2D(2,1) pattern); this is shown in Figure 7. This time, the sliding direction is along the column. If a third node (15) fails, then there is no space left for the 1D sliding (top row of Figure 7). This situation can be avoided by sliding along the column direction after the second failure (middle row of Figure 7).

The number of nodes below which a third failure cannot be handled by the 1D sliding method is the product of the number of slidings in each direction, when the thickness of the spare node set is one. Thus, it is not a good idea to evenly distribute the sliding directions; instead, they should be as uneven as possible. Even when this is done, however, the 1D sliding method may be limited to three failures (bottom row in Figure 7).

The relation between the maximum number of message collisions and the number of failed nodes with the 2D(2,1) spare node allocation pattern can be expressed as shown below. Note that there may be cases in which this method cannot handle more than three node failures

$$C_{\max} = 2 + F_n$$

2.2.3. 2D sliding, 3 sliding, . . . , qD sliding. In the 2D sliding method, the rows and columns of the node space are shifted by one unit to empty the row or column of the failed node (bottom panel of Figure 5). This 2D sliding method can handle only one node failure with the 2D(1,1) pattern or two node failures with the 2D(2,1) pattern.

If the network has a higher-dimensional Cartesian topology than 2D, then the 3D or higher-order sliding can take

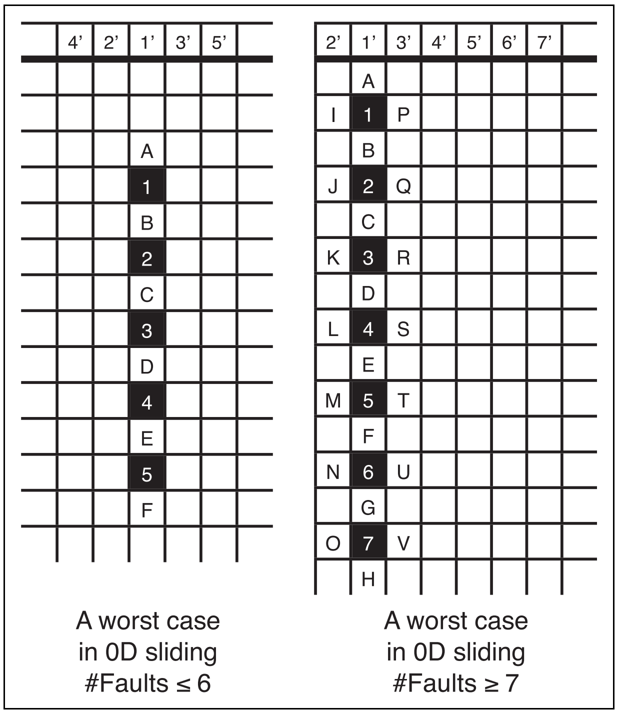
4' 2' 1' 3' 5'
A 1 B 2 C 3 D 4 E 5 F
2' 1' 3' 4' 5' 6' 7'
A I 1 P B J 2 Q C K 3 R D L 4 S E M 5 T F N 6 U G O 7 V H
A worst case in 0D sliding #Faults ≤ 6
A worst case in 0D sliding #Faults ≥ 7

Figure 6. Worst-case scenarios for 0D sliding.

place in the same way. The highest degree of a sliding method is equal to the number of dimensions of a Cartesian network and this sliding method can handle up to the number of dimensions.

With the XY routing, the messages pass orthogonally through the vacant rows or columns. All message routes are the same as they were before the failure. Thus, unlike the 0D and 1D sliding methods, although the hop counts are increased by one, message congestion can be avoided. Further, this behavior is independent of the communication pattern of the application.

2.3. Comparison of proposed methods

Figure 8 shows the number of possible message collisions versus the number of failed nodes for the 0D sliding method with the 2D(1,1) and 2D(2,1) spare node allocation patterns, 1D sliding with the 2D(2,1) pattern, and 2D sliding with the 2D(2,1) pattern. These numbers are obtained by our developed simulation program with which every possible combination of node failures is simulated so that the number of message collisions in a 5P-stencil communication is counted at every link and the highest number of message collisions is reported. In this simulation, it is assumed that four messages of 5P-stencil communication are sent simultaneously.

As already described, the number of possible message collisions with 0D sliding with the 2D(1,1) allocation pattern for a given number of failed nodes depends on the network topology (mesh or torus) when the number of faults is greater than six (upper left-hand panel in the figure). With 2D(2,1) case, up to five failures are simulated. It is possible to handle more number of failures with the 0D sliding method; however, the exponential growth of failure combinations was the obstacle for us to simulate more.

The 1D sliding method with the 2D(2,1) spare node allocation pattern can handle up to three failures perfectly. More number of failures can be handled when the failures happen at some specific locations. This is shown as a dashed line in Figure 8.

The 1D sliding method can handle no more than the number of spare nodes minus one, since the spare node at the corner of the 2D(2,1) allocation cannot be used. The 2D sliding with 2D(2,1) can handle only two failures as described before.

The sliding method has a good characteristic where node migration can take place in a pipeline fashion. Therefore, the time to migrate nodes can be independent ($O(1)$, assuming the amount of data to be migrated are the same over nodes) from the number of migrating nodes.

2.3.1. Hybrid method.

The substitution methods described so far are independent and can be applied in a combined way. Figure 9 shows an example of a hybrid method. The first and second failures are handled by using the 2D sliding method (left-hand and middle panels), and the third failure is handled by using the 1D sliding method (right-hand panel). In this way, message collisions can be avoided even up to two failures, and the job can survive even with a greater number of failures.

Thus, a hybrid sliding method can be expressed with the set of sliding methods and the order of their applications. As described in Section 2.3, the higher-order sliding methods incur lower message collisions but the number of failure able to handle is smaller. Thus, the order of sliding methods to be applied in a hybrid method should be a descending order of the degree of sliding method. Hereinafter, a hybrid method will be expressed as "hybrid(2D + 1D + 0D)," for example, meaning 2D sliding is applied whenever it is possible, then 1D sliding method is applied, and finally 0D sliding method is applied. Since 0D sliding method can be applied in any circumstances, any hybrid method should have 0D sliding method as a last resort. In this article, a hybrid sliding method combining all possible sliding methods with the descending order of degrees is also denoted as "hybrid(all)" for short. Another hybrid method combining all possible methods except X is denoted as "hybrid(-X)."

In the next section, some simulation results will be shown followed by an evaluation section on the K computer, BG/Q, and TSUBAME 2.5 (Endo et al., 2014). One may argue that the numbers (percentages) of failed nodes simulated and evaluated in this article are too many and not realistic. However, those simulations and evaluations are done to reveal the behavior and characteristics of the proposed substitution methods. We believe that the research on how to utilize spare nodes is a new frontier of fault mitigation.

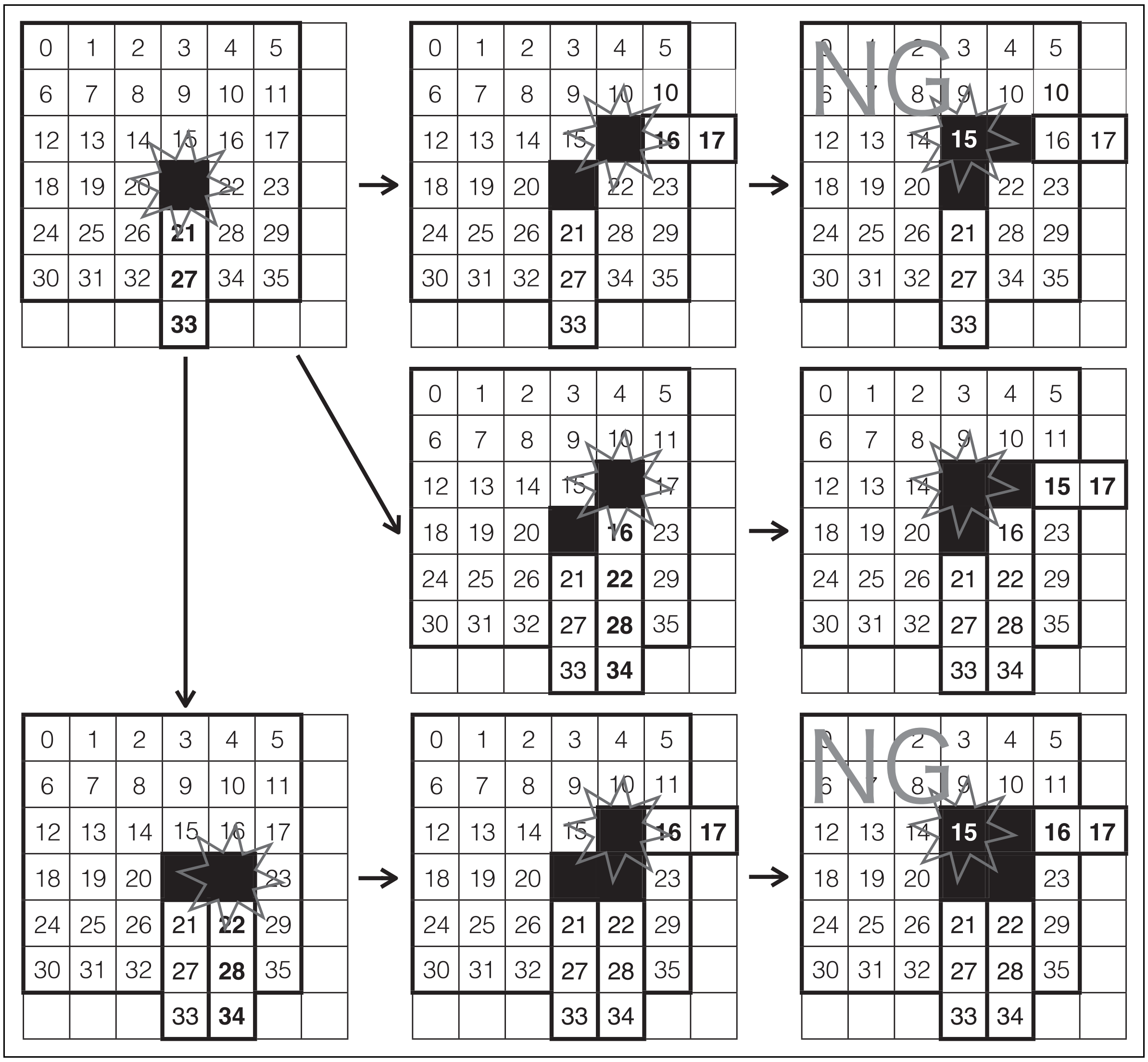

Figure 7. Example of 1D sliding.

3. Simulation

We developed a simulator to imitate the proposed sliding methods on a stencil communication with a Cartesian network and counted the maximum number of possible message collisions. We made the assumption that all stencil messages are sent on all stencil dimensions simultaneously. Thus, the counted message collisions is the theoretical maximum and the actual message collisions on a real machine can be less than the simulated numbers because of the skew of sending messages to all dimensions. The skeleton of the simulation algorithm is described below:

(1) choose one alive node randomly,
(2) mark the chosen node as *failed* and apply a sliding method if the chosen node is not a spare node,
(3) repeat above procedure until the number of failed nodes reaches the specified number of failure,
(4) save the node-rank mapping information to a file,
(5) simulate a stencil communication and increment *msg_count* of all links on the paths from the source nodes to the destination nodes, and
(6) report the maximum value of *msg_count* at all network links.

Since the number of combination of failed nodes is the factorial of the number of failed nodes, it is impossible to simulate all possible cases especially when the number of nodes is large. Instead of having the exhaustive search, the simulation results shown in this section are obtained with random sampling. The resulting node-rank mapping pattern is saved into a file so that we can evaluate it on the real platform allowing the assessment of the quality of the evaluation with real data on a realistic scenario.

We chose the node spaces, 100 × 100 (2D), 12 × 12 × 12 (3D), and 24 × 24 × 24 (3D). We chose the base number of 12 in 3D cases because the minimum unit of the K computer network is 12 (called Tofu unit) to make the

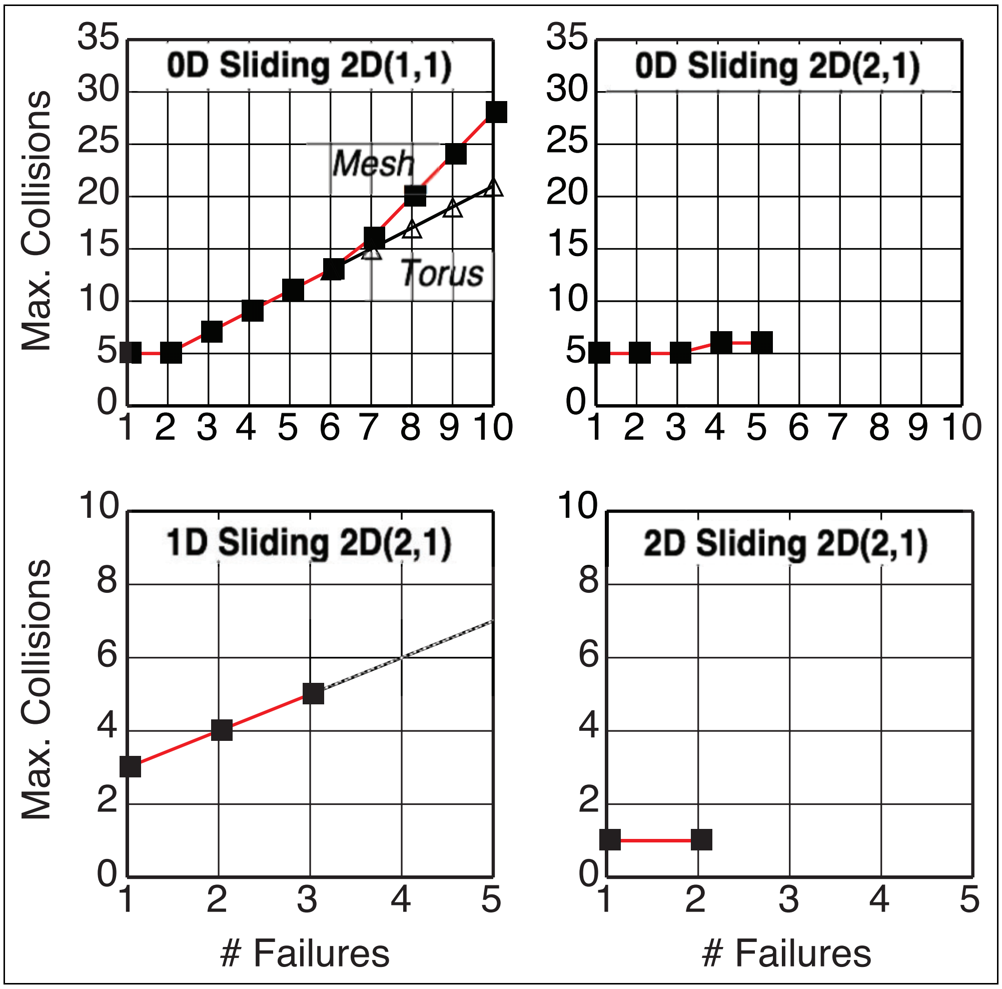

Figure 8. Comparison of 0D, 1D, and 2D sliding (5P-stencil, worst cases with exhaustive search).

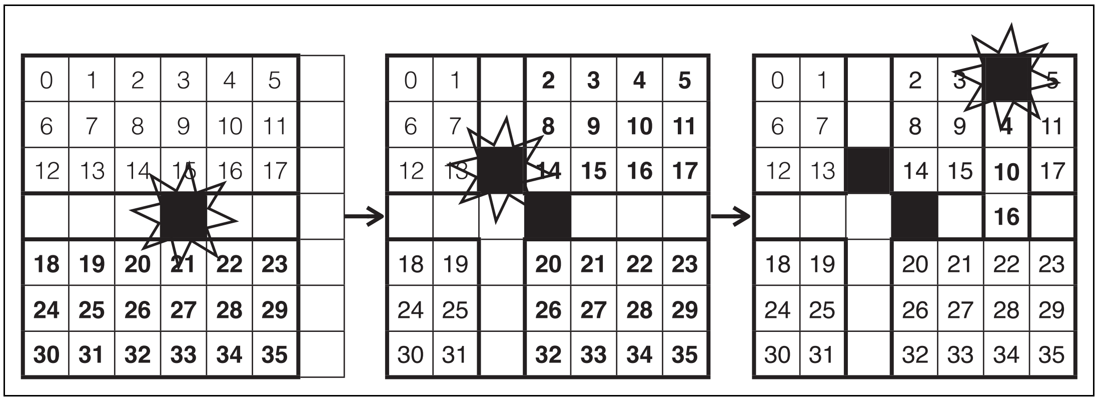

Figure 9. Example of hybrid(2D + 1D) sliding.

simulation results comparable with the actual evaluation on the K computer in the next section.

Figures 10 to 12 show this simulation results on 2D network (5P-stencil, 100 × 100, 2D(2,1) spare node allocation, 3,340,000 random cases), 3D network (7P-stencil, 12 × 12 × 12, 3D(2,1) spare node allocation, 3,686,400 random cases), and 3D network (7P-stencil, 24 × 24 × 24, 3D(2,1) spare node allocation, 3,686,400 random cases), respectively. Each graph compares hybrid sliding method (three thick lines, worst, average, and best from top to down) and 0D sliding (three thin lines, similarly, worst, average, and best from top to down) method. "Best" in the legend means the lowest number of message collisions, "Worst" means the largest number of collisions and "Average" means the average of all cases.

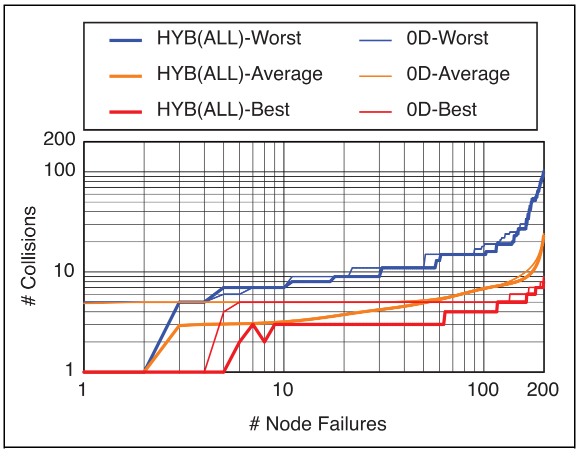

Figure 10. Hybrid(all) versus 0D, 2D network (100 × 100), 2D(2,1) spare nodes.

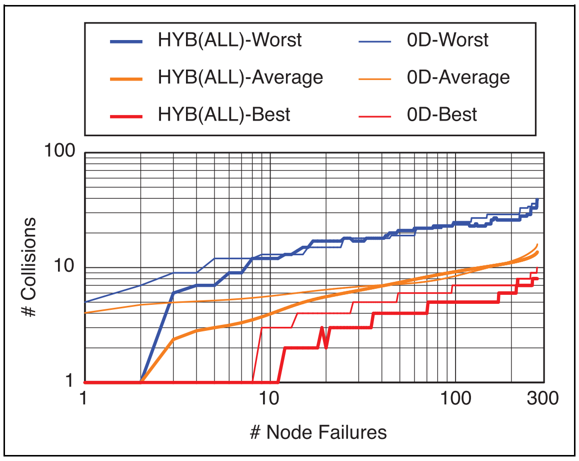

Figure 11. Hybrid(all) versus 0D, 3D network (12 × 12 × 12), 3D(2,1) spare nodes.

In the graphs in Figures 10 and 11, hybrid sliding method outperforms 0D sliding method in terms of best, average, and worst cases in the smaller number of node failures. In Figure 12, however, the hybrid method outperforms 0D sliding not so much as in the cases of 100 × 100 and 12 × 12 × 12. Comparing the worst numbers, 0D sliding is better in the range of the number of node failures between 10 and 155. Comparing the average numbers, 0D sliding is better in the range of the number of node failures between 248 and 740.

It is obvious that stencil communication matches with Cartesian topology and no message collisions happen if the degree of the network is equal to or higher than the degree of the stencil communication. When a sliding method higher than 0D takes place, the network links connecting the nodes adjacent to the *sliding planes* can result in message congestion. Here, *sliding plane* is defined as the planes which surround the sliding nodes, except 0D sliding. The number of nodes (or the size of area of the plane) adjacent to the 1D sliding plane is smaller than that of 2D sliding. Although the maximum number of message collisions of

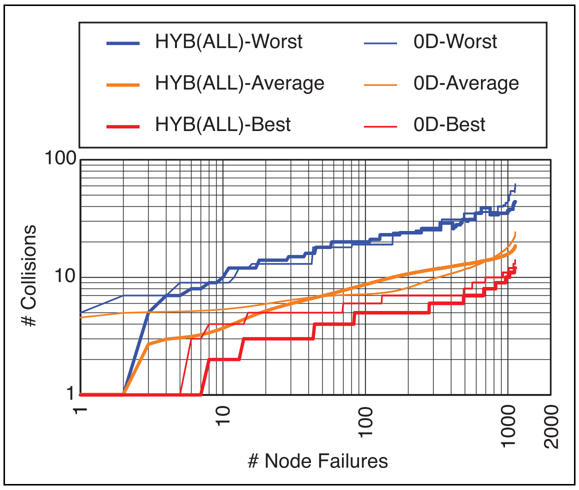

Figure 12. Hybrid(all) versus 0D, 3D network (24 × 24 × 24), 3D(2,1) spare nodes.

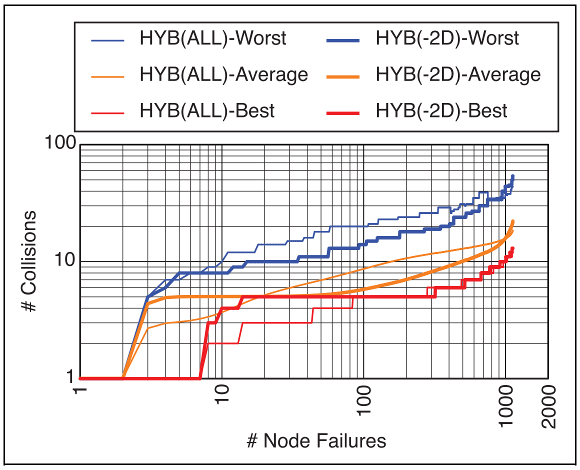

Figure 13. Hybrid(all) versus hybrid(-2D), 3D network (24 × 24 × 24), 3D(2,1) spare nodes.

1D sliding and 2D sliding are the same, the number of network links having collisions caused by the 2D sliding is bigger than that of 1D. So, the possibility of adding more message collision(s) to the link(s) gets higher. From this viewpoint, 1D sliding might be better than 2D sliding. Based on this idea, hybrid(-2D) or hybrid(3D + 1D + 0D) might outperform hybrid(all).

Figure 13 shows the simulation results of hybrid(all), already shown in Figure 12, and hybrid(-2D) to compare. In this figure, the lines of hybrid(-2D) are thick. When attention is paid to the average lines, hybrid(all) outperforms hybrid(-2D) at the rages at the leftmost part, from 1 to 18, and the rightmost part, from 993 to 1128 which is the number of spare nodes. Middle part excepting those ranges, however, hybrid(-2D) performs very well.

Figure 14 shows the rates of which sliding methods are used in the sampling set. Figure 15 shows the accumulated selection rate. As shown in these figures, the first two node failures are substituted by using the 3D sliding method.

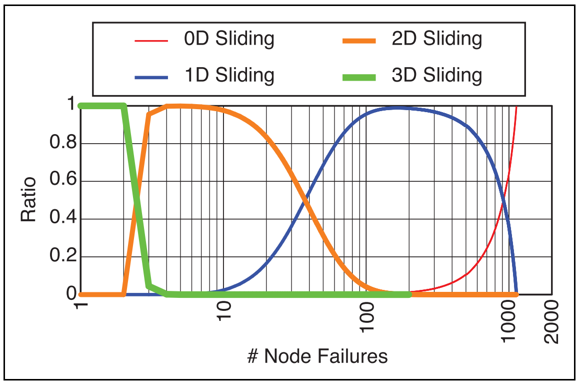

Figure 14. Selection rate of hybrid(all) sliding, 3D network (24 × 24 × 24), 3D(2,1) spare node allocation.

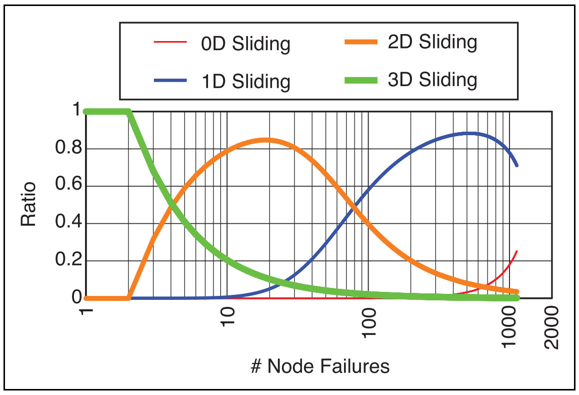

Figure 15. Accumulated selection rate of hybrid(all) sliding, 3D network (24 × 24 × 24), 3D(2,1) spare node allocation.

Because spare node allocation is 3D(2,1), two is the maximum number of 3D sliding methods. 2D sliding method dominates until having 38 node failures. Then, 1D sliding method takes over until 915 node failures. Finally, 0D sliding method takes the rest. As shown in Figure 15, when all the spare nodes are substituted, 0D, 1D, and 2D sliding methods occupy 25%, 71%, and 5%, respectively.

4. Evaluations on K, BG/Q, and TSUBAME 2.5

In this section, the sliding methods described so far are evaluated by using the actual supercomputers; the K computer JUQUEEN (Stephan, 2012), a BG/Q machine, and TSUBAME 2.5 listed in Table 1. This experimental campaign will characterize the difference between the theoretical analysis and observed practical consequences.

The stencil communication is simulated by replicating the usual communication pattern in an MPI-based stencil: non-blocking communications with the neighbors (`MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` functions); all send and receive buffers are contiguous and no MPI-derived datatype is used. Processes were placed such that a single MPI process per compute node was used. In addition to the point-to-point communications, we simulate widely used collective operations, `MPI_Barrier` and `MPI_Allreduce`, which were measured by repeating the collective call. The message size of the stencil communication was set to 4 MiB and the message size of allreduce communication is set to 64 KiB. A node failure was simulated by excluding a node from the computation. It was almost impossible to have a real node failure since all the machines we used were in operation and a real node failure, even if it could be simulated by making it offline, may affect the jobs of the other users running at that time. On the K computer and BG/Q, the MPI implementations provided by the vendors were used. Supposedly, those collective operations are tuned for their network topology and mapping. This would reveal how the spare node allocation and sliding substitution would affect the collective performance. On TSUBAME 2.5, MVAPICH2, an open-source MPI implementation, was used.

We have focused our analytical effort on the maximum number of message collisions, which has the implicit assumption that all messages are sent from nodes simultaneously, thereby always resulting in collisions if their path follows the same link. However, the number of message that can be sent simultaneously is dependent on network hardware features (i.e. the number of Direct Memory Access (DMAs)). When the actual number of simultaneous sends is only one, for example, the number of messages injected into the network is decreased and the number of collisions is also reduced.

The rank mapping after the node failure(s) used in this evaluation are the same as the ones used in the simulation in the previous section. We measured the latency of those communication operations and the slowdown ratio is calculated based on the latency having spare nodes without having any node failure.

$$R^i = L_{\text{no\_sub}}/L^i_{\text{sub}}$$

where L^i_{sub} is the measured communication latency after ith substitution, $L_{\text{no\_sub}}$ is the latency without having any node failure but having spare node set, and the R^i is the slowdown ratio at that time.

The proposed sliding methods have been explained and discussed by using a 2D Cartesian network; however, the actual physical network can be more complex, having five or more number of dimensions, as shown in Table 1. Even if users require their jobs to run in 2D node spaces, those 2D node spaces are folded to fit in the actual network topologies. On the K computer, any 2D Cartesian node planes are mapped to the 6D Tofu network so that the neighbor relationship of the 2D or 3D Cartesian topology can be preserved. On the BG/Q system, the node-rank mapping is the user's responsibility. To preserve the neighbor relationship of the 2D or 3D Cartesian topology, "snake-like pattern" is recommended (IBM, 2013a). On TSUBAME 2.5, the physical node space is one-dimensional, dare to say. Anyhow, the mapping or folding of users' topologies to fit into a physical network topology may affect the communication performance in different ways discussed so far.

Table 2 lists some characteristics of the K computer, BG/Q, and TSUBAME 2.5. The K computer has four DMA engines and up to four messages can be sent simultaneously. BG/Q has 11 FIFOs and up to 10 messages can be sent simultaneously. On the other hand, the network of TSUBAME 2.5 is Infiniband (Infiniband Trade Association, n.d.). TSUBAME has two Infiniband Host Channel Adapters (HCAs) on a node, one of them is used in this article to avoid the interference with the other jobs. So, TSUBAME 2.5, in this evaluation, can send only one message at a time.

The rightmost two columns of this table show the ratios to send multiple messages simultaneously, four messages with 5P-stencil, six messages for 7P-stencil, based on the time to send one message. These values, except TSUBAME 2.5, are measured by our program. In theory, five message collisions, for example, means the communication time gets slower five times. On the K computer, only three times slower communication time was observed because simultaneous four message sending takes 1.7 times of the time of sending one message ($3 \approx 5/1.7$) (Hori et al., 2015). One possible reason to explain this slowness (1.7 with 5P-stencil and 3.7 with 7P-stencil) is the insufficient bandwidth between the memory and the network controller chip.

In the following subsection, the evaluation results of the K computer and BG/Q are shown, followed by the evaluation results of TSUBAME 2.5. This is because they have Cartesian network topologies while TSUBAME 2.5 has FatTree network topology. And the behavior of the K and BG/Q is very different from that of TSUBAME 2.5.

4.1. Evaluations on K and BG/Q

4.1.1. Stencil communication. Figure 16 shows the simulation results on the 3D ($12 \times 12 \times 12$) network and the evaluation results of the K computer with the $12 \times 12 \times 12$ node allocation. Spare nodes are allocated in the way of 2D(2,1). The upper graph in this figure shows and compares the results using hybrid(all) sliding method, and the lower graph shows the results of using hybrid(-2D) sliding method. The 768 failure patterns (the set of failed nodes) are chosen from the worst cases in the simulation.

As shown in both graphs, the average lines observed on the K computer is almost always better than the result of the simulations. This is considered as that the actual network degree of the K computer (6D) is higher than the degree of the simulation (3D). The links provided by the additional dimensions give the paths to bypass resulting lower message collisions. Comparing the best lines, the simulation outperforms in the rage of less than or equal to 10 node failures. This is because of the sampling effect, the number of simulation cases is much bigger than that of evaluation and the best cases found in the simulation could not be

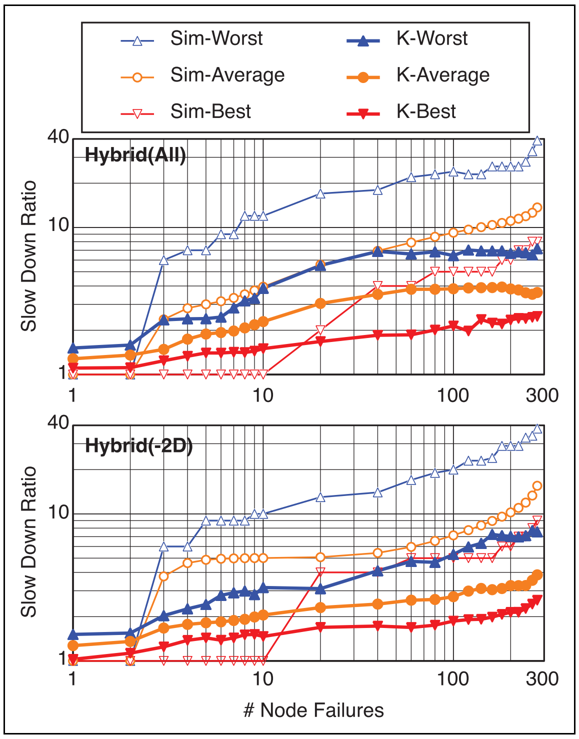

Figure 16. Stencil communication on K ($12 \times 12 \times 12$), 3D(2,1) spare node allocation.

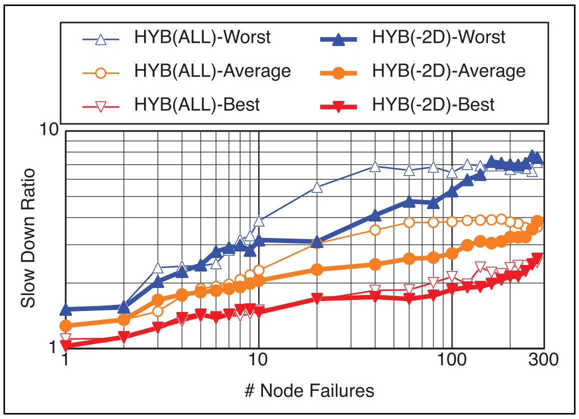

Figure 17. Hybrid(all) versus hybrid(-2D) on K ($12 \times 12 \times 12$), 3D(2,1) spare node allocation.

found in the evaluations. The same situation happens on the worst cases.

To compare hybrid(all) and hybrid(-2D), Figure 17 shows the evaluation results of them (the same data used in Figure 16). The effect of hybrid(-2D) shown in this figure is very similar to the one found in Figure 13.

Figure 18 shows the simulation results on the 3D ($16 \times 8 \times 8$) network and the evaluation on BG/Q computer with the $16 \times 8 \times 8$ node allocation. Spare nodes are allocated in the way of 2D(2,1). As in the previous graphs, the upper graph in this figure shows and compares the results using hybrid(all) sliding method, and the lower graph shows the results of using hybrid(-2D) sliding method. Here again, the

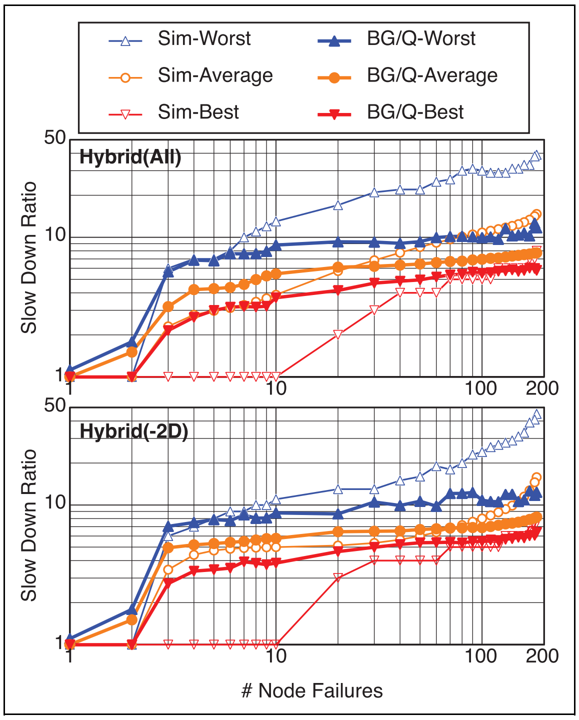

Figure 18. Stencil communication on BG/Q (16 × 8 × 8), 3D(2,1) spare node allocation. BG/Q: Blue Gene/Q.

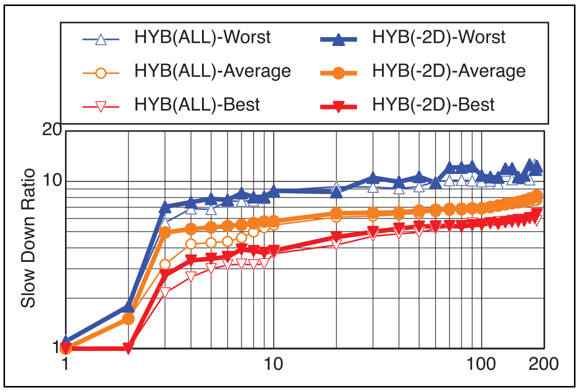

Figure 19. Hybrid(all) versus hybrid(-2D) on BG/Q (16 × 8 × 8), 3D(2,1) spare node allocation. BG/Q: Blue Gene/Q.

768 failure patterns (the set of failed nodes) are chosen from the worst cases in the simulation.

Comparing the BG/Q results and the K computer results (Figure 16), the differences between the evaluation and the simulation in the range of less than or equal to 10 node failures are very small. This might come from the fact that the network degree of the BG/Q (5) is less than the degree of the K computer (6) and/or that the network topology of BG/Q and the K computer are different. The BG/Q network topology is full five dimensions, while the K computer, the last three dimensions (A, B, C links out of X, Y, Z, A, B, C) are used to form a 2 × 3 × 2 subnetwork unit (Tofu unit).

Figure 19 shows the evaluations of hybrid(all) and hybrid(-2D) (the same data used in Figure 18). Unlike the

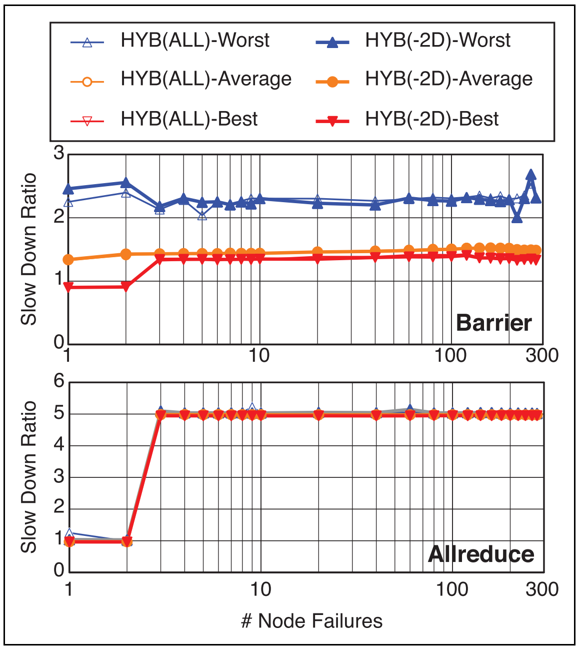

Figure 20. Barrier and allreduce on K (12 × 12 × 12), 3D(2,1) spare node allocation.

case on the K computer (Figure 17), the difference between hybrid(all) and hybrid(-2D) is very small.

4.1.2. Collective communication.

Up to now, the peer-to-peer (P2P) communication performance in 5P-stencil communication pattern has been the primary focus. In this subsection, we will extend to the case of collective communication performance. The communication patterns of collective communications are more varied than the stencil pattern, thereby providing a wider insight about less regular P2P communication patterns as well.

On the K computer, the Tofu network supports hardware barrier. The other various collective communications are tuned so that the best performance can be obtained based on the Tofu network topology and characteristics. The tuning of collective protocols is also very important for the Cray's Gemini network (Peña et al., 2013). However, it is very difficult to predetermine optimized collective protocols for any possible set of node failures.

In order to use the tuned collective protocol for the Tofu network, each MPI collective communication has some conditions for the physical shape of the communicator. Some of the conditions come from the special protocol tuned for the Tofu network, and the others come from implementation issues. When a substitution is made for a failed node, one or more of these conditions cannot be met and generic algorithms are used. Thus, the performance of the collective communication can degrade much more than that of the stencil communication, because the special tuned protocols cannot be applied in addition to the collision issue.

Figure 20 shows the relative performance of barrier (upper graph) and allreduce (lower graph) collective

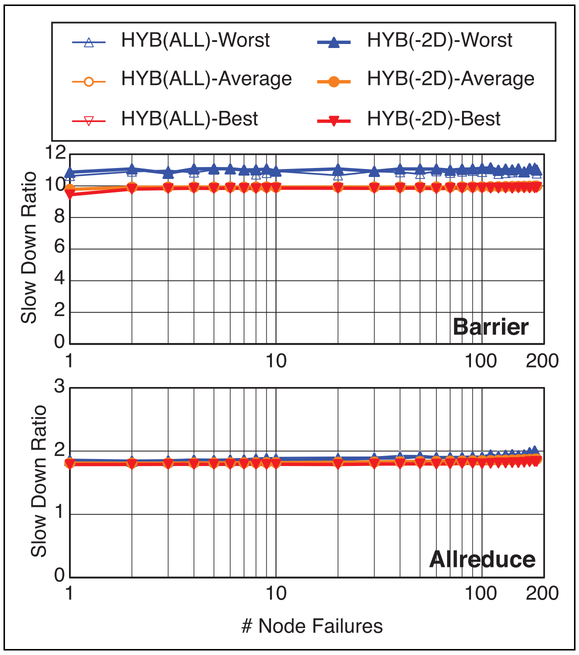

Figure 21. Barrier and allreduce on BG/Q (16 × 8 × 8), 3D(2,1) spare node allocation. BG/Q: Blue Gene/Q.

communications based on the performance of no substitutions are made. Nodes of the evaluation job are allocated in 3D (12 × 12 × 12) and the sampling set is the same as the evaluation of stencil on the K computer in the previous subsection.

In the allreduce case, the slowdown cannot be seen with one node failure and two node failures. In this evaluation, the hybrid sliding methods are used and spare nodes are allocated in 3D(2,1), so 3D sliding method can be applied up to two node failures. The sliding method having the largest degree happens with no message collision. And this property of the sliding method with the largest degree does not break the conditions where the optimized allreduce protocol on the K computer can be applied.

Figure 21 shows the barrier and allreduce performance on BG/Q. We found that the collective performance on the node set having spare nodes is slower than the cases without having any spare node; 8.2 times slower with the barrier operation and 1.8 times slower with the allreduce operation. Such slowdown cannot be seen on the K computer.

To make sure, we evaluated the barrier performance without the snake-like pattern mapping. When the spare nodes were allocated on one specific physical dimension out of the five dimensions of the BG/Q network, such barrier performance degradation could not be seen. However, when one node was excluded from `MPI_COMM_WORLD`, then the barrier performance was slowed down to one-tenth. Apparently, the barrier operation on BG/Q is optimized for `MPI_COMM_WORLD`. Thus, the best way is to apply the snake-like pattern only to the nodes not reserved for spares.

Unlike the K computer collective cases, the barrier and allreduce performance of BG/Q is quite stable over the number of node failures independently from the hybrid methods. The 3D sliding method which is effective up to two node failures in this case does not help to improve the situation.

Table 2. K, BG/Q, and TSUBAME 2.5.

			Ratio to 1 message sending	
Machine Name	Topo.	# DMAs	5P-stencil	7P-stencil
K	6D Cart.	4	1.7	3.7
BG/Q (Chen et al., 2011)	5D Cart.	10	1.1	1.3
TSUBAME	FatTree	1[a]	4[b]	6[b]

BG/Q: Blue Gene/Q.
[a]TSUBAME 2.5 has two IB networks but only one of them is used to avoid interference with the other jobs.
[b]Estimated values.

4.2. Evaluations on TSUBAME 2.5

We ran the same evaluation programs used in the previous subsection on TSUBAME 2.5, 7 × 7 × 7 node space with 2D(2,1) spare node allocation. Unlike the K and BG/Q cases, we could not see any significant slowdown. The communication performance is almost constant within the range of 1.0–1.2 over the sliding methods and the number of node failures, without obvious correlation. To make sure of this phenomenon, we additionally evaluated the sliding methods with random node-rank mapping. As far as we tried, no obvious performance degradation can be seen.

One possible reason for this phenomenon is that the network topology of TSUBAME 2.5 is two-stage FatTree. The node space is expressed in one-dimensional way for the sake of convenience. However, there is no significance on the node numbers of the nodes connecting to the same "edge" switch. Any changes on node-rank mapping on those nodes have no effect.

The other factor for this phenomenon is that the network of TSUBAME 2.5 is Infiniband. Indeed, TSUBAME 2.5 has two network sets so that the communication bandwidth can be doubled by utilizing them in the multi-rail way (Liu et al., 2004). One of the network sets is also used for Lustre file system and it is likely to have I/O traffics of the other jobs. So we decided to use the other network set to avoid the interference with the other jobs. Each Infiniband HCA has one DMA engine and, unlike the K computer and BG/Q, only one message can be sent from an HCA at a time. In a stencil communication, in theory, multiple messages can be sent simultaneously, however, TSUBAME 2.5 has the capability of sending only one message in this case. This situation is very different from the K computer able to send up to 4 messages and BG/Q able to send up to 10 messages simultaneously (Table 2). This leads to have less injected messages in the network and to have less chance to have message collisions. This may happen also with the collective communications.

5. Related work

Shadow replication proposes another replication scheme for fault handling to meet service level agreement (SLA) (Mills et al., 2014). Their unique feature is to minimize the additional power consumption by having replication (shadow) processes. The authors mind the cloud service providers, SLA, and the cost for power consumption. They assume that the speed and power consumption of application execution running as the shadow processes can be controlled by using Dynamic Voltage and Frequency Scaling (DVFS). In HPC, unlike cloud computing, applications are supposed to run as fast as they can. Further, most supercomputer users do not make SLA contracts. Having any sort of replication process consumes twice hardware resource. In our proposed spare node utilization model, the additional resource for fault mitigation is far less than twice as shown in Figure 3.

Ferreira et al. (2011) indicated that dual hardware redundancy while utilizing only 50% of the hardware resource, might be under some assumptions more efficient than the traditional checkpoint and restart method in Exascale systems. These redundancies can be thought of as spare nodes. The difference is that the redundant nodes are *hotter*-standby than the hot-standby nodes waiting for the intermediate computational results. The spare nodes can be substituted for the failed nodes, and they can almost immediately take over the computations.

Domke et al. (2014) showed the difference in communication performance between the presence or absence of network failure (link or switch) over different network topologies and routing algorithms. They analyzed the communication performance degradation when network links or switches failed; this was done by simulation using TSUBAME 2.0. In the K computer, the Tofu direct network has redundant routes to bypass failed nodes. However, a job is aborted and resubmitted by the operating system if it uses a failed part. In this work we focus on node failures rather than network failures. There is a long way to go until we reach the goal where any kind of failures, node and/or network, can be mitigated.

Brown et al. (2015) proposed a visualizing system of message traffics in a communication network and they succeeded to identify hot spots. In their case study using the samplesort program running on TSUBAME 2.5, 5% performance gain was obtained by avoiding the hot spots which they discovered by using their tool. Conversely speaking, their paper reveals that finding an optimal node-rank mapping to level hot spots according to network topology and communication pattern of an application is not an easy task.

Fang et al. (2015) argued about the shrinking and non-shrinking post-recovery strategies. Although they use the spare node for the non-shrinking strategy, they did not propose any strategy on how to allocate a spare node set, how to select a spare node from the spare node set, nor how the failed node is substituted. Further, their evaluation was done only with BG/Q, whereas we evaluated three different supercomputers, the K, BG/Q, and TSUBAME 2.5, and revealed that the communication performance degradation depends on the network. They focused on the increased hop counts after the recovery, but we focused on the number of message collisions. As already described, the modern high performance network exhibits very small latency per hop (below 100 ns). The point here is that the larger the number of hops, the higher the possibility of having message collisions, and therefore an increased impact on the communication performance. It should be noted that they reported the non-shrinking strategy by using spare nodes as superior, in terms of efficiency, to the shrinking strategy if failure rate is high. We believe the non-shrinking strategy proposed in this article would be accepted by users in terms of ease of programming and performance.

As reported so far, our proposed sliding methods are to reduce the additional latency for the communication after the failed node substitution. There are ongoing works to reduce the communication time by introducing new algorithms to improve the communication time in parallel programs. Communication avoiding algorithms (Demmel et al., 2012) for linear algebra and temporal blocking algorithm for stencil computation (Muranushi and Makino, 2015) are the examples of this approach. These algorithms could improve both of the program execution time without having any failed nodes and the execution time after having failed node substitution. An application that succeeds to hide communication latency perfectly may fail to hide the increased latency due to the spare node substitutions. Thus, it is important to explore the way to minimize the latency introduced by spare node substitutions.

ULFM proposed by Bland et al. (2013) supports both shrinking and non-shrinking recovery. Upon the occurrence of a failure, instead of aborting the application like in legacy MPI, ULFM produces an error code from the impacted communication routines at surviving processes. Surviving processes may then continue to operate using point-to-point routine between themselves, or interrupt the communication pattern of the application with `MPIX_Comm_revoke` and rebuild fully functional communicators excluding failed processes with `MPIX_Comm_shrink`. When deploying this shrinking recovery model with ULFM, the underlying implementation has no notion of active and spare ranks. From the ULFM perspective, all ranks are application processes, with the same features. Our work expands on this basic framework by adding at the user level an initial spare allocation strategy as well as a substitution strategy that avoids communication hot spots in the recovered application. The non-shrinking recovery in ULFM builds on top of the previously described shrinking model. Once a new communicator fit for issuing collective operation has been reconstructed with the shrink routine, spawning additional processes can be achieved with traditional MPI-2 dynamic process management routines (e.g. `MPI_Comm_spawn`). The ULFM spawn accepts nonstandard arguments to finely select where to allocate additional processes; it is possible to select, from the application, which node will host the supplementary processes. It is important to note however that the default policy in ULFM is a simple round-robin among the available slots in the existing allocation. The user remains in charge of the decision-making for advanced placement policy making even in non-shrinking cases.

Fenix proposed by Gamell et al. (2014) is a framework to automate the procedure to substitute failed node(s) with spare node(s) and to resume execution from a checkpoint. Laguna et al. (2014) proposed *Reinit* API which is an extension of existing MPI for bulk synchronous applications to survive from node failure. Fenix and Reinit provide easier-to-use API than ULFM and support a non-shrinking strategy by assuming a spare node set. Both works depend on the `MPI_Comm_spawn*` functions to substitute failed nodes. This means that the selection of a spare node and definition of a spare node set depend on the system and choosing a spare node according to a specific substitution strategy is out of control of applications. Further, both did not take into account the possibility of communication performance degradation after the recovery. In contrast, this article focused on the possibility of such performance degradation and proposed sliding substitution methods to minimize the degradation. Our proposed sliding methods can be integrated into such frameworks to improve the performance after recovery.

6. Discussion

6.1. Node utilization in a multijob environment

The possibility that a job has a failed node is proportional to the number of nodes assigned to the job and execution time. Thus, the number of spare nodes must also be proportional to the number of nodes assigned and execution time. Therefore, the number of spare nodes allocated by the proposed method may be excessive when only a small number of nodes are required by a given job. Ideally, the curves shown in Figure 3 would be a horizontal line at the height determined by node failure rate, if the execution times are the same.

Figure 22 shows a countermeasure for this. Large jobs should have a higher-order spare node allocation method, and smaller jobs should have a lower-order method; this will allow the spare node percentage to approximate a horizontal line. In the example shown in Figure 22, the spare node percentage is kept in the range from 2% to 5% by using a combination of the 3D(2,1), 3D(2,1), and 3D(1,1) methods.

6.2. User-level versus system-level substitutions

So far in this article, we have considered methods in which the spare nodes are allocated by the job. We would like to develop a framework that uses something like ULFM and that framework automatically replaces faulty nodes with spare nodes, so that users do not need to be concerned with how failures are handled. GVR supporting versioned data

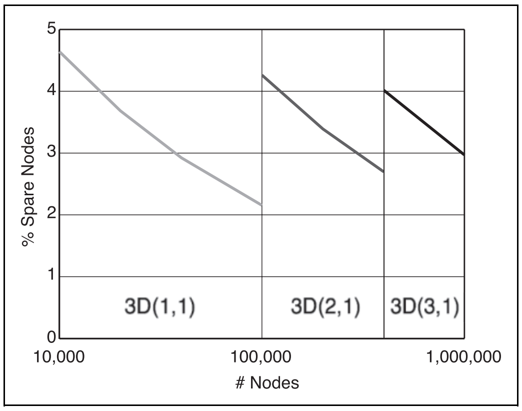

Figure 22. Combinations of spare node allocation methods.

backup and providing higher abstraction than that of ULFM might be able to support the proposed spare node substitution.

When spare nodes are allocated and substitutions are determined by user programs, this is called as *user-level substitution*; when this is done at a lower system software level, it is called *system-level substitution*. With the user-level substitution, the user program is also invoked at each spare node, and it waits in hot-standby mode for the data to migrate from the failed node. This means that calling `MPI_Comm_spawn` is not required. On the other hand, system-level substitution can reduce the percentage of spare nodes, because spare nodes can be shared by several jobs. For example, spare nodes can be allocated at the boundaries of jobs, and these can be used to replace failed nodes on every side of the job boundaries. However, it is not possible to have spare nodes on hot-standby, as with user-level substitution. If the spare nodes are not adjacent to the job in which they are needed, this can result in uncontrollable message collisions with other jobs, and unexpected communication performance degradation.

6.3. Job resubmission versus fault mitigation

One may argue that a job can be aborted and then resubmitted using a checkpoint, instead of mitigating the fault. In this way, the problem of utilizing spare nodes and the degradation of communication performance, described above, can be avoided. Job resubmission, however, may incur a long turnaround time, especially when the system is heavily loaded, and user-level fault mitigation techniques, such as those described in Davies et al. (2011), cannot be utilized. When considering which is better, there are many aspects to be considered. In this article, we considered only the effect on communication performance. It is still an open question if it is better to resubmit a job or mitigate the fault.

6.4. Worm-eaten node space

So far in this article, a job is allocated with a node space where there is no node failure at the beginning. Usually, when a node failure happens, the failed node is to be physically replaced with a new healthy node. To replace the failed node, firstly this node is unplugged from a rack and or chassis and then sanity node is plugged in. When a node is unplugged from a rack or chassis and if its network is a direct network, then the network switch (router) associated with the node is also gone. Thus, unplugging a node simulates a network switch failure. When a switch failure happens, network routing must be changed to bypass the failed switch and this may affect the other running jobs. So, the physical node replacement cannot take place soon after the node failure happens.

It is expected that Mean Time Between Failure (MTBF) is increasing in the future. And there will be the case where node failure happens much more frequently. Thus, the number of node failures will increase within the interval of repairing nodes. This may result in the situation where large jobs might be allocated with a "worm-eaten" node space at the beginning, instead of having a healthy contiguous node space. In this case, node substitution methods described in this article and/or algorithms to find an optimal node-rank mapping can be applied at the beginning of job execution. Therefore we believe the research on node substitution and the algorithm to find (sub)optimal node-rank mapping will be very important.

7. Summary and future work

In this article, we considered methods for allocating spare nodes and replacing failed nodes in jobs whose rank-node mapping is critical to performance. We compared these methods in terms of communication performance following substitutions. The substitution methods are 0D, 1D, 2D, and higher sliding methods. In the stencil communication, the higher the order of the sliding method, the fewer message collisions but more failure distributions are unrecoverable for lack of spares. Thus, a combination of these methods would seem to be the best strategy. We also extended the evaluation to widely used collective operations.

We also revealed that such performance degradation after the substitution may depend on network characteristics and node-rank mapping. If the high failure rate becomes reality, a network might be designed in such a way to minimize the performance degradation after the substitution.

Utilizing spare nodes influences various fields in high performance computer design, hardware and software. As shown in the evaluations, a network topology plays an important role. It is expected that the communication performance degradation found in the substitutions can be relaxed by having a dynamic routing. The mapping of applications' communication patterns and optimizations of collective communication patterns to fit in the available network topology becomes very difficult with the presence of failed node substitutions. Because there is no regular pattern where node failure happens and the number of node failure patterns is explosive. Therefore the assumption to have spare nodes has a significant impact on hardware and software design.

The research on this failed node substitution with spare nodes has just begun. We will continue investigating on this research topic.

A part of the results in this article is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science (HPCI Project ID `hp150240`).

Acknowledgements

The authors thank Dr Norbert Attig at Jülich Supercomputing Center for allowing them access to the JUQUEEN platform. They also thank Dr Franck Cappello at Argonne National Laboratory for his useful comments.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was partially supported by the CREST project of the Japan Science and Technology Agency (JST).

References

Ajima Y, Sumimoto S and Shimizu T (2009) Tofu: a 6D mesh/torus interconnect for exascale computers. *Computer* 42(11): 36–40.

Bland W, Bouteiller A, Herault T, et al. (2013) Post-failure recovery of MPI communication capability: design and rationale. *International Journal of High Performance Computing Applications* 27(3): 244–254.

Brown K, Domke J and Matsuoka S (2015) Hardware-centric analysis of network performance for MPI applications. In: *2015 IEEE 21st international conference on parallel and distributed systems (ICPADS)*, Melbourne, Victoria, Australia, 14–17 December 2015, pp. 692–699. IEEE. DOI: 10.1109/ICPADS.2015.92.

Cappello F, Geist A, Gropp W, et al. (2014) Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations* 1(1): 5–28.

Chen D, Eisley NA, Heidelberger P, et al. (2011) The IBM Blue Gene/Q interconnection network and message unit. In: *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, SC '11*, Seattle, WA, USA, 12–18 November 2011, pp. 1–10. IEEE. DOI: 10.1145/2063384.2063419.

Chen Y, Plank JS and Li K (1997) CLIP: a checkpointing tool for message-passing parallel programs. In: *Proceedings of the 1997 ACM/IEEE conference on supercomputing (SC '97)*, pp. 1–11. ACM.

Chen Z and Dongarra J (2008) Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems* 19(12): 1628–1641.

Chien A, Balaji P, Beckman P, et al. (2015) Versioned distributed arrays for resilience in scientific applications: global view resilience. *Procedia Computer Science* 51: 29–38.

Davies T, Karlsson C, Liu H, et al. (2011) High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: *Proceedings of the international conference on supercomputing, ICS '11*, Tucson, AZ, USA, May 2011, pp. 162–171. New York, NY: ACM.

Demmel J, Grigori L, Hoemmen M, et al. (2012) Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34(1): 206–239.

Di Martino C, Kalbarczyk Z, Iyer R, et al. (2014) Lessons learned from the analysis of system failures at petascale: the case of blue waters. In: *2014 44th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, Atlanta, GA, USA, 23–26 June 2014, pp. 610–621. IEEE. DOI: 10.1109/DSN.2014.62.

Domke J, Hoefler T and Matsuoka S (2014) Fail-in-place network design: interaction between topology, routing algorithm and failures. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC '14*, New Orleans, LA, USA, November 2014, pp. 597–608. Piscataway, NJ: IEEE.

Endo T, Nukada A and Matsuoka S (2014) TSUBAME-KFC: a modern liquid submersion cooling prototype towards exascale becoming the greenest supercomputer in the world. In: *2014 20th IEEE international conference on parallel and distributed systems (ICPADS)*, Hsinchu, Taiwan, 16–19 December 2014, pp. 360–367. IEEE. DOI: 10.1109/PADSW.2014.7097829.

Fang A, Fujita H and Chien AA (2015) Towards understanding post- recovery efficiency for shrinking and non-shrinking recovery. In: *Euro-Par 2015: Parallel Processing Workshops* (eds S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Goémez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander), Cham, 2015, pp. 656–668. Springer International Publishing.

Ferreira J, Stearley JH, Laros III, et al. (2011) Evaluating the viability of process replication reliability for exascale systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, 1–12. ACM. Doi: 10.1145/2063384.2063443.

Fujita H, Dun N, Fang A, et al. (2014) Using global view resilience (GVR) to add resilience to exascale applications. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC '14*, New Orleans, LA, USA, November 2014. Piscataway, NJ: IEEE.

Gamell M, Katz DS, Kolla H, et al. (2014) Exploring automatic, online failure recovery for scientific applications at extreme scales. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC '14*, New Orleans, LA, USA, November 2014, pp. 895–906. Piscataway, NJ: IEEE.

Hori A, Yoshinaga K, Herault T, et al. (2015) Sliding substitution of failed nodes. In: *Proceedings of the 22nd European MPI users' group meeting, EuroMPI '15*, Bordeaux, France, September 2015, pp. 1–10. New York, NY: ACM.

IBM (2013a) *IBM System Blue Gene Solution: Blue Gene/Q Application Development*, 2nd ed. IBM.

IBM (2013b) *IBM System Blue Gene Solution: Blue Gene/Q System Administration*, 2nd ed. IBM.

Infiniband Trade Association. InfiniBand (n.d.). Available at: http://www.infinibandta.org/ (accessed 2 January 2020).

Laguna I, Gamblin T, Mohror K, et al. (2014) *A global exception fault tolerance model for MPI*. Technical Report LLNL-CONF-659977. Livermore, CA: Lawrence Livermore National Lab.

Liu J, Vishnu A and Panda DK (2004) Building multirail InfiniBand clusters: MPI-level design and performance evaluation. In: *Proceedings of the 2004 ACM/IEEE conference on supercomputing (SC 04)*, Pittsburgh, PA, USA, 6–12 November 2004, p. 33. IEEE. DOI: 10.1109/SC.2004.15.

Message Passing Interface Forum (2012) MPI: A Message-Passing Interface Standard Version 3.0. Available at: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (accessed 2 January 2020).

Mills B, Znati T and Melhem R (2014) Shadow computing: an energy-aware fault tolerant computing model. In: *2014 International conference on computing, networking and communications (ICNC)*, Honolulu, HI, USA, 3–6 February 2014, pp. 73–77. IEEE. DOI: 10.1109/ICCNC.2014.6785308.

Muranushi T and Makino J (2015) Optimal temporal blocking for stencil computation. *Procedia Computer Science* 51: 1303–1312.

Peña AJ, Carvalho RGC, Dinan J, et al. (2013) Analysis of topology-dependent MPI performance on Gemini networks. In: *Proceedings of the 20th European MPI users' group meeting, EuroMPI '13*, Madrid, Spain, September 2013, pp. 61–66. New York, NY: ACM. DOI: 10.1145/2488551.2488564.

Sato K, Maruyama N, Mohror K, et al. (2012) Design and modeling of a non-blocking checkpointing system. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis, SC '12*, Salt Lake City, UT, USA, November 2012, pp. 1–10. Washington, DC: IEEE Computer Society Press. Available at: http://dl.acm.org/citation.cfm?id=2388996.2389022.

Stephan M (2012) JUQUEEN: Blue Gene/Q—system architecture. Available at: http://www.training.prace-ri.eu/uploads/tx_pracetmo/JUQUEENSystemArchitecture.pdf (accessed 8 March 2016).

Strohmaier E, Dongarra J, Simon H, et al. (2015) TOP500 Supercomputer Sites. Available at: http://www.top500.org/ (accessed 8 March 2016).

Sumimoto S (2012) The MPI communication library for K computer: its design and implementation. Invited talk at *EuroMPI 2012*, Vienna. Available at: http://www.par.univie.ac.at/conference/eurompi2012/docs/s9t1.pdf (accessed 8 March 2016).

Takefusa A, Ikegami T, Nakada H, et al. (2014) Scalable and highly available fault resilient programming middleware for exascale computing. In: *Proceedings of the international conference for high performance computing, networking, storage* *and analysis, SC '14*, New Orleans, LA, USA, November 2014.

Teranishi K and Heroux MA (2014) Toward local failure local recovery resilience model using MPI-ULFM. In: *Proceedings of the 21st European MPI users' group meeting, EuroMPI/ASIA '14*, Kyoto, Japan, September 2014, pp. 51–56. New York, NY: ACM. DOI: 10.1145/2642769.2642774.

Zhang W, Hou L, Wang J, et al. (2009) Comparison research between XY and odd-even routing algorithm of a 2-dimension 3×3 mesh topology network-on-chip. In: *Proceedings of the 2009 WRI global congress on intelligent systems, GCIS '09*, Xiamen, China, 19–21 May 2009, Vol. 03, pp. 329–333. Washington, DC: IEEE Computer Society.

Author biographies

Atsushi Hori received Ph.D. from University of Tokyo in 1999. He is a senior scientist of Riken Center for Computational Science, Japan. His research interests include parallel processing, parallel hardware, and runtime system.

Kazumi Yoshinaga is a systems engineer working for eF-4 Co., Ltd., Japan. He worked as a Postdoctoral Researcher at RIKEN AICS from 2013 to 2016. He received his B.E., M.E. and Ph.D. degrees in Computer Science and Systems Engineering from Kyushu Institute of Technology.

Thomas Herault is a research director at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests include fault tolerance, distributed algorithms, parallel programming paradigms, and performance modelling and optimizations. He focuses on bridging the gap between theoretical distributed systems and high-performance computing as it is practiced.

Aurélien Bouteiller received his Ph.D. from the University of Paris in 2006. He is currently a Research Director at the Innovative Computing Laboratory, the University of Tennessee, Knoxville. He focuses on improving performance and reliability of distributed systems with research in communication and scheduling for many-core, accelerated clusters, and in containing the influence of failures on scientific computation.

George Bosilca is a research assistant professor and Adjunct Assistant Professor at the Innovative Computing Laboratory at The University of Tennessee, Knoxville. His research interests evolve around programming paradigms for distributed non-reliable platforms, message passing and other approaches to deal with the scalability, heterogeneity and resiliency at any scale and in any settings.

Yutaka Ishikawa is the leader of Fugaku supercomputer development project that deploys the next Japanese flagship supercomputer at Riken Center for Computational Science, Japan. From 2002 to 2014, he was a professor at the University Tokyo. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership, Japan.