Sliding Substitution of Failed Nodes

Atsushi Hori
RIKEN AICS
ahori@riken.jp

Kazumi Yoshinaga
RIKEN AICS
kazumi.yoshinaga@riken.jp

Thomas Herault
The University of Tennessee
Innovative Computing Lab.
herault@icl.utk.edu

Aurélien Bouteiller
The University of Tennessee
Innovative Computing Lab.
bouteill@icl.utk.edu

George Bosilca
The University of Tennessee
Innovative Computing Lab.
bosilca@icl.utk.edu

Yutaka Ishikawa
RIKEN AICS
yutaka.ishikawa@riken.jp

ABSTRACT

This paper 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 node-rank mapping may be destroyed. Therefore, these questions must be answered in a way that minimizes the degradation of communication performance.

In this paper, several spare-node allocation and node-substitution methods will be proposed, analyzed, and compared in terms of communication performance following the substitution. It will be shown that when a failure occurs, the peer-to-peer (P2P) communication performance on the K computer can be slowed by a factor of three and collective performance can be cut in half. On BG/Q, P2P performance can be slowed by a factor of five and collective performance can be slowed by a factor of ten. However, those numbers can be reduced by using an appropriate substitution method.

CCS Concepts

•Networks → Network experimentation; •Computer systems organization → Redundancy;

Keywords

fault tolerance, fault mitigation, spare node, communication performance

1. INTRODUCTION

With the fault rate increasing on high-end supercomputers, the topic of fault tolerance has been gathering attention[3], and jobs are being aborted due to system errors[7]. 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 of checkpoint and restart, but the most notable one is [4]).

With the increase in size of parallel applications, the amount of I/O needed for checkpoint/restart begun to be problematic. A lot of research is currently going on on techniques to reduce the checkpoint amount in order to alleviate the I/O issue. One of the possible approaches is [14]. 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 greatly reduced, at the cost of only some additional programing by the user.

Davies *et al.* presented a method that allows a user program to be fault-tolerant without using checkpointing[5]. In this technique, the parity to recover the lost data can be embedded into an 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 programing paradigm and the execution environment, users could write application handling faults in the most optimal way. The Message Passing Interface (MPI) is the most widely used communication library, and its specifications are well defined[12]. 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[2]. 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 programing[19]. Local Failure Local Recovery (LFLR) is another fault-tolerant framework[20], and it covers a wider range of programing models than are supported by Falanx. Both

EuroMPI '15, September 21 - 23, 2015, Bordeaux , France

ISBN 978-1-4503-3795-3/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2802658.2802670

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 (PGAS) programing[9, 22].

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.

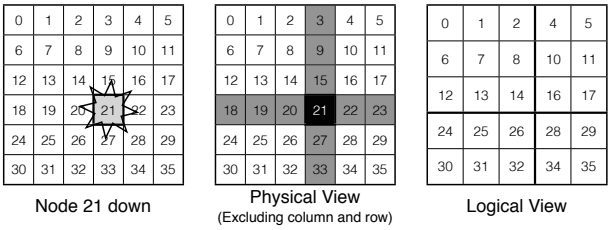

Figure 1: Example of node failure and recovery

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 remain 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.

This paper presents the results of our investigations into these issues. We propose several methods for using spare nodes to replace faulty ones. The proposed methods are discussed and compared from the viewpoint of communication performance degradation. The primary contribution of this paper is a comparison of methods for allocating spare nodes and for substituting them for faulty nodes, while focusing on the degradation of communication performance.

2. USING SPARE NODES

For the remainder of this paper, we will assume that the networks being considered have a multidimensional Cartesian (mesh and/or torus) topology. We make this assumption because four of the top five machines have networks with this topology (as listed on the TOP500 Super Computer Site[17], November 2014); see Table 1.

Table 1: Network topologies in the Top500 list[17]

Top500 rank	Name	# Cores	Topology
1	Tianhe-2	3,120K	FatTree
2	Titan (Cray XK7)	561K	3D Torus
3	Sequoia (BG/Q)	1,571K	5D Torus/Mesh
4	The K computer	705K	6D Torus/Mesh
5	Mira (BG/Q)	786K	5D Torus/Mesh
8	JUQUEEN (BG/Q)	459K	5D Torus/Mesh

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 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 turn-around time, ease of user programing, and the framework that needs to be developed. Unfortunately, almost no research has been done on this topic, so in this paper, we will focus primarily on the communication performance.

Throughout this paper, 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[8] (see also Section 4). The Tofu network, which is used by the K computer, uses redundant links to detour around failed nodes[18]. 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

For simplicity, we will consider only 2D networks. In higher-dimensional networks, that is, with a higher order of XY routing[21], if messages are routed in the dimension order, then they are eventually routed to a plane consisting of the last two dimensions. Thus, a discussion of a 2D network is meaningful.

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 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.

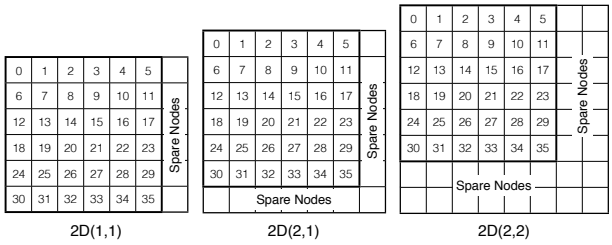

Figure 2: Patterns for allocation of spare nodes

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.

$$R_{2D(2,2)} = 1 - (N^{1/2} - 2)^2/N$$

Where N is the number nodes. In the more general $qD(r, s)$ case, the percentage of spare nodes can be expressed as follows.

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

Here, $r \leq q$ and $s < N^q$. 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 for various numbers of nodes and patterns of 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 5.1.

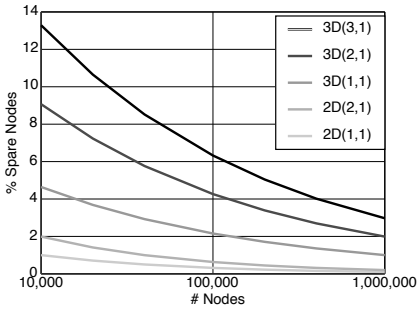

Figure 3: Percentage of spare nodes

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 Section 2.2, 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 twelve nodes. When a user submits an 11x11x11 3D job, for example, it may be scheduled to have 12x12x12 nodes. This results in 3D(3,1) 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[11]. On a Cray XK/7, jobs are allocated to 4 blocks[13]. 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.

2.2 Substitution of a Spare Node for a Faulty Node

Communication performance degradation can be observed because 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, 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.

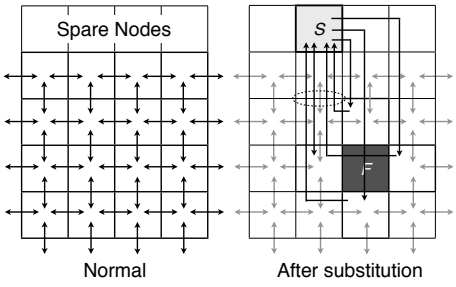

Figure 4: Message collisions

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. 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 non-periodic 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 eleven messages in the network links between 1' and A (these are shown in the white boxes): these ten 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,

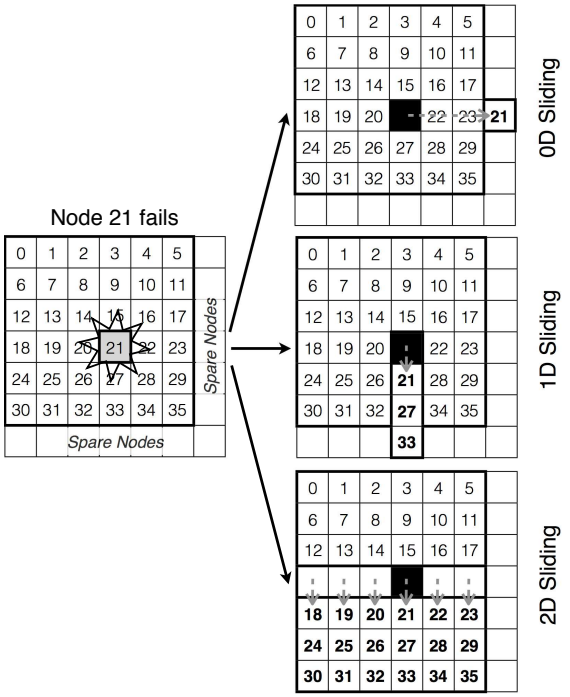

Figure 5: Substitution methods for faulty nodes

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 non-periodic 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 sixteen messages, since each node sends four 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.

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

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. See Section 2.3.

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 5D-stencil communication pattern is increased by only one. This is much smaller than occurs with the 0D sliding method.

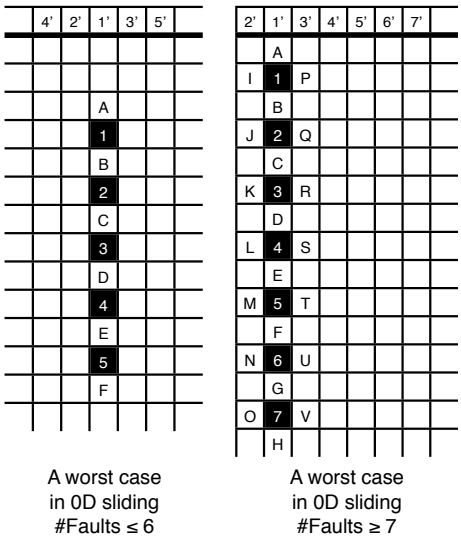

Figure 6: Worst-case scenarios for 0D sliding

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

The 1D sliding method can be modified so that it can handle four or more failures. The left-hand panel of Figure 8 is the same as the bottom right-hand panel of Figure 7. When the fourth node (15) fails, nodes 9, 10, and 11 are slid to the right to make room above node 15, and node 15 is moved one space upward (right-hand panel of Figure 8). This method is called the *1D+ sliding* method.

2.2.4 2D sliding, 3D 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, then the 3D or higher-order sliding can take place in the same way.

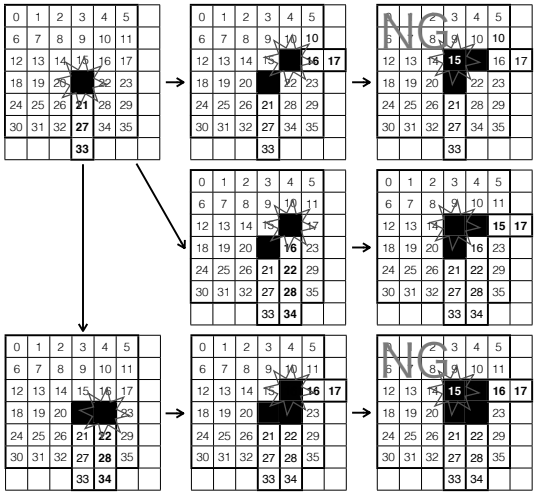

Figure 7: Example of 1D sliding

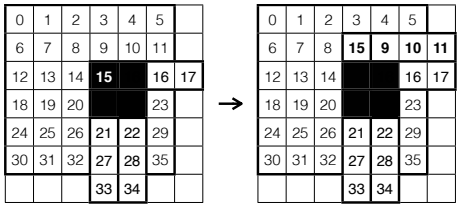

Figure 8: 1D+ sliding

With 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 9 shows histograms of the cases having the largest message collisions in any possible combinations of a failed node and a spare node. The 5P-stencil communication pattern, no periodic boundaries, was simulated. The 0D (left graph) and 1D sliding (right graph) methods are compared with the following node allocation patterns: 10x10, 20x20, 40x40, 80x80, and 160x160 (mesh topology). The Y axes shows the normalized frequency of the combinations of failed node and spare node.

As can easily be seen, the larger the number of nodes, the higher the frequency of high message collisions. This is because fewer message collisions happen when a failed node is 1) close to the spare nodes; 2) on the side of the node grid; or 3) on the boundary of the 5P-stencil. Thus, when the network topology is a torus and/or the 5P-stencil computation has periodic boundaries, then the worst case will occur more frequently. When the number of nodes is very

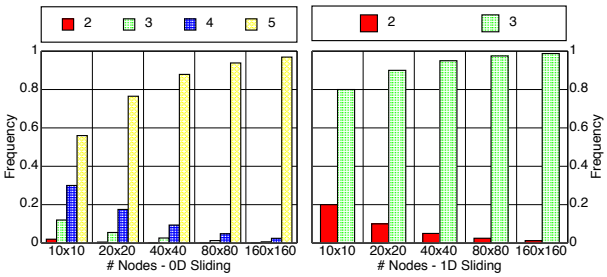

Figure 9: Histogram for number of collisions (5P-stencil, one node failure, simulated)

large, the worst-case scenario for message collisions happens in most cases.

Figure 10 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.

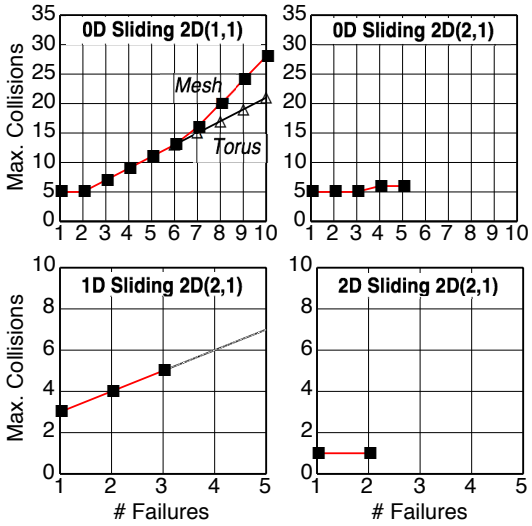

Figure 10: Comparison of 0D, 1D, and 2D sliding (5P-stencil, worst cases)

As already described in Section 2.2.1, 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 5 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 10.

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.

Hybrid method

The substitution methods described so far are independent and can be applied simultaneously. Figure 11 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 with two failures, and the job can survive even with a greater number of failures.

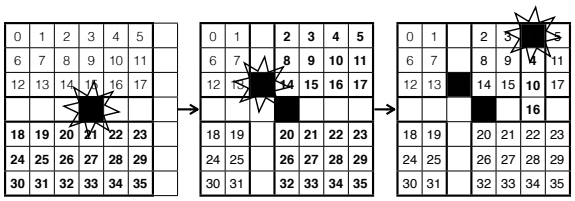

Figure 11: Example of hybrid sliding

If stencil program users are willing to have the load redistribution shown in Figure 1, then the nodes left vacant by the handling of the first node failure can be used as spare nodes, and the 0D, 1D, 1D+, and 2D sliding methods can be applied to subsequent failures. With cost to the user for extra programing effort to adjust the load distribution, the node utilization problem described in Section 2.1 can be avoided.

As shown in Figure 11, the node-rank mapping will be lost as the number of failed nodes increases. Thus far, we have considered the number of possible message collisions for each method separately, but we have not considered the number of collisions when they are mixed; thus, we should consider the possibility of one or more hot spots for collisions in the network. To avoid this situation, when the disorder reaches a given level, the nodes should be reordered so that any hot spots will be removed. Unfortunately, we have not yet been successful in developing such an algorithm.

3. EVALUATION

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 5 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[10]. Anyhow, the mapping or folding of users' topologies to fit into a physical network topologies may affect the communication performance in different ways discussed so far.

Also in this paper, 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 (like the number of DMAs). When the maximum number of simultaneous sends is one, for example, the number of collisions is reduced.

In this section, the sliding methods described so far are evaluated by using the actual supercomputers: the K computer and JUQUEEN[16], a BG/Q machine listed in Table 1. The snake-like pattern appropriate for BG/Q computers is employed on JUQUEEN. This experimental campaign will characterize the difference between the theoretical analysis and observed practical consequences.

3.1 5P-Stencil Communication

In the 2 failure cases in this subsection, all possible combinations of 2 node failures are simulated. Communication performance degradation is measured with an in-house 5P stencil communication benchmark.

In most figures in this section, the worst times are shown by the upper horizontal bars, the best times are at shown by the lower horizontal bars, and the average values are shown by the middle horizontal bars. The Y-axes are the relative times, compared to the cases without having any failed nodes (thus, bigger is worse).

Figure 12 shows the relative communication performance of 5P-stencil communication on the K computer. The number of nodes allocated for the job is 24×24 (576) and spare nodes are allocated in the 2D(2,1) way. In the 1D+ sliding method, only the 2 failure cases are shown because its substitution is the same as the one of the 1D sliding method with having only one failure. The substitutions of 2 failed nodes in the 1D+ sliding method are also the same with the 1D sliding method when the spare nodes are allocated in the 2D(2,1). So the 1D+ sliding method was evaluated as if it is allocated 24×23 nodes withe the 2D(1,1) spare node allocation. Message size varies between 256KiB to 4Mib.

In the 1D+ sliding method, in most cases the substitution after 2 faults are similar to the 1D sliding method. We outline the difference between the 1D and 1D+ sliding methods by presenting separately the cases where those methods actually result in a different rank-node mapping (in the graph noted 1D+ only).

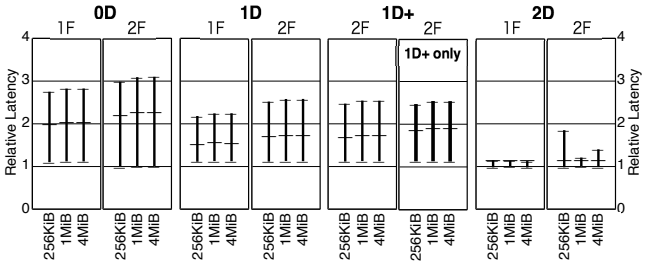

Figure 12: 5P Stencil - the K computer

So far, it has been assumed that the message sending in a stencil communication happens simultaneously. However, jitter[1] affects this assumption. Assume that two messages, A and B having the same length, arrive at the same network link at the same time, and the message A can go through before the message B. The message B must wait until the whole message A goes through the link. This results in that the message B will have twice the latency.

Similarly, when the message B arrives at a link junction in the middle of the message A going through the link, the latency of the message B is 1.5 times larger compared with the case without collision. Thus, when jitter is large or message sizeis small, the actual average latency affected by message collisions get smaller.

In the 0D sliding cases shown by this graph, the worst latencies here is almost three times worse than the latency when there are no faulty nodes. As discussed above, there are at most five messages colliding. However, the Tofu network of the K computer allows us to send four messages in different directions at the same time[15], but this takes from 1.64 (256MiB message) to 1.75 (4MiB message) times longer than it does to send a single message. Thus, a lag of $5/1.7 \approx 3$ is observed. On BG/Q, contrastingly, the 4-way simultaneous message sending takes from 1.12 (256KiB message) to 1.03 (1MiB and 4MiB message) times larger than the time for sending only one message.

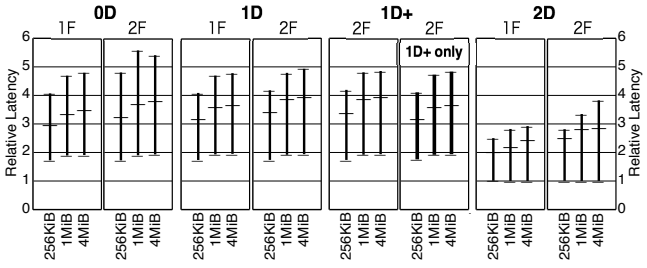

Figure 13: 5P Stencil - BG/Q

Figure 13 shows the BG/Q results of the same evaluations as in the above K computer cases, except that the number of nodes allocated for the job is 16×32 (512) and 15 $(16-1)$ spare nodes are used in the 1D+ sliding method cases. In the one node failure cases, no significant difference can be seen among the 0D, 1D and 1D+ methods. In the two node failure cases, the 0D method performs worse than the others.

Comparing the K computer cases and the BG/Q cases, BG/Q is more sensitive to the message size and less sensitive to the sliding method, than the cases of the K computer. The communication performance degradations of the K computer look less than those of BG/Q, however, this seems to come from the performance of simultaneous message sending of the K computer. In the cases with the 0D sliding method, the latency ratios of the K computer multiplied by 1.6 or 1.7 are close to the values of BG/Q.

Remarkably, the 2D sliding method performs very well on both the K computer and BG/Q. Comparing the graphs of 1D+ sliding method and the graphs denoted as "1D+ only" on both the K computer and BG/Q, the differences between them are small.

Figure 14 shows the observed communication performance degradation with one node failure while varying the total number of nodes measured on the K computer. Here, the message size is set to 4MiB, a 5P-stencil communication pattern with the 0D sliding method is used. The goal is to verify that the communication overhead is independent from the number of nodes.

The collisions resulting from node substitution(s) in the

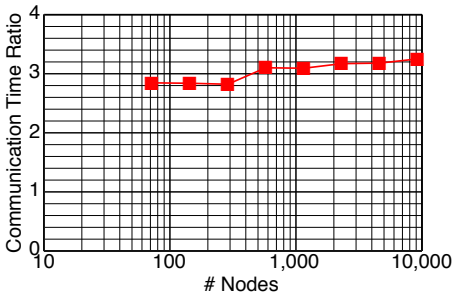

Figure 14: Worst-case 5P-stencil communication time with the 0D sliding method on the K computer

5P-stencil communication pattern are described in Subsection 2.2. How the message collisions happen with the node substitution(s) in the 5P-stencil communication is described. Theoretically, those collision patterns are independent from the number of nodes (when the number of nodes are large enough). In a stencil computation, all communications happen simultaneously. When a message collides with another message, then the message is stopped due to the collision. When the collision is over and the stopped message starts moving again, the other messages which did not collide with any other messages are already received by the corresponding receiver nodes and thus there is no messages left to collide with in the network. Therefor any message are blocked by collisions at most once in a stencil communication. This is the reason why the communication performance degradation of a stencil communication pattern is independent from the number of nodes.

3.2 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 that 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 characteristics[18]. The tuning of collective protocols is also very important for the Cray's Gemini network[13]. However, it is very difficult to predetermine optimized collective protocols for any possible set of node failures.

In order to tune collective protocol for the Tofu netwotk, 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 15 shows the barrier and allreduce performances (message size was 64KiB) on the K computer, with one and two failed nodes, replaced using the 0D, 1D, 1D+, and 2D sliding methods. The number of nodes and the evaluation condition for the 1D+ sliding are the same as in the previous subsection of 5P-stencil evaluation.

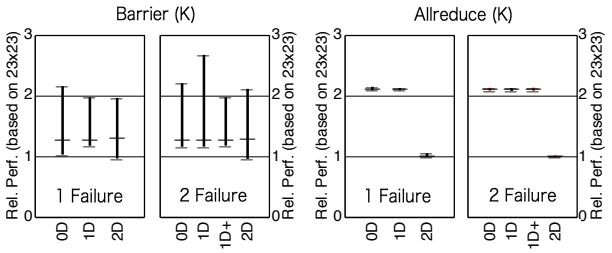

Figure 15: Barrier and Allreduce - K (24x24 nodes)

In the worst case of the barrier evaluation graphs in Figure 15, 2.68 times slower than the cases of no failures with the 1D sliding method. The average values (middle bars) are in the range of 1.26-1.29. One can note that the variance in the allreduce cases is much smaller than the cases of the barrier evaluations and the performance degradations of the 2D sliding method are insignificant.

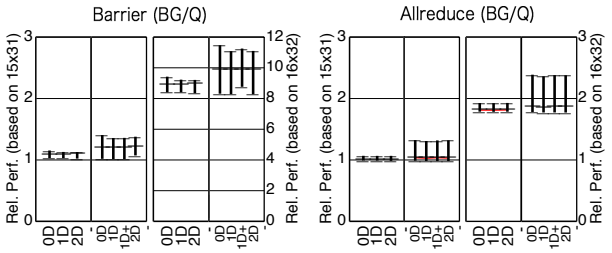

Figure 16: Barrier and Allreduce - BG/Q (16x32 nodes)

Figure 16 shows the barrier and allreduce performance on JUQUEEN. We found that the collective performance on the node set having a spare node set 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 on BG/Q. Such slowdown cannot be seen on the K computer. There are two sets of graphs, one is based on the collective performance with the node set excluding the spare nodes (left graph pairs) and the other is based on the performance with the node set having no spare nodes (right graph pairs) to be fair. 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 5 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. Thus, the best way is to allocate spare nodes according to the network topology and then apply the snake-like pattern to the node space without spare nodes. However, in this way, the behavior of the message collisions discussed so far can be quite different.

Comparing the graphs of the K computer and the graphs of BG/Q, BG/Q is more sensitive to the number of faults.

Especially, there cannot be seen almost any slowdown in the allreduce cases with the 2D sliding method on the K computer, while no significant differences over the various sliding methods can be seen on BG/Q.

4. RELATED WORK

Ferreira *et al.* 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 This 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.* showed the difference in communication performance between the presence or absence of network failure (link or switch) over different network topologies and routing algorithms[8]. 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.

5. DISCUSSION

5.1 Node Utilization in a Multijob Environment

Most supercomputers use a batch scheduling system, in which many jobs run simultaneously. The spare node percentages shown in Figure 3 are for individuals jobs, not systems. If a machine has one million nodes and 100 jobs are running (for simplicity, assume that these jobs each require 10,000 nodes), then the overhead cost of spare nodes can exceed 10%.

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 17 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 17, the spare node percentage is kept in the range from 2% to 5% by using a combination of the 3D(3,1), 3D(2,1), and 3D(1,1) methods.

5.2 User-level vs. System-level Substitutions

So far in this paper, 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

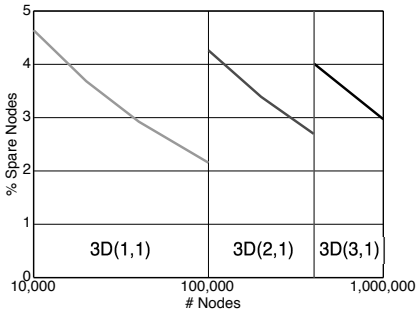

Figure 17: Combinations of spare node allocation methods

that framework replaces faulty nodes with spare nodes, so that users do not need to be concerned with how failures are handled. 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 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 both sides of the boundary. 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.

5.3 Job Resubmission vs. 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 [6], cannot be utilized. When considering which is better, there are many aspects to be considered. In this paper, 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. SUMMARY AND FUTURE WORK

In this paper, 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, 1D+, 2D, and higher sliding. In 5P-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.

When replacing a failed node with a spare node, the spare node must be integrated into the computations. In 1D, 1D+ or 2D sliding, the computations on these nodes are migrated. We are planning to develop a framework to hide the complexity of allocating and utilizing spare nodes. The technical issues and actual overhead cost of doing so will be reported when the development is completed.

Acknowledgment

We thank Dr. Norbert Attig at Jülich Supercomputing Center for allowing us access to the JUQUEEN platform. We also thank Dr. Franck Cappello, at Argonne National Laboratory, for his useful comments. This research is partially supported by the CREST project of the Japan Science and Technology Agency (JST).

7. REFERENCES

[1] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–12, Sept 2006.

[2] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[3] F. Cappello, A. Geist, W. D. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 Update. *Supercomputing Frontiers and Innovations*, 1:1–28, 2014.

[4] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE Conference on SuperComputing (SC'97)*, pages 1–11, Nov 1997.

[5] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.*, 19(12):1628–1641, Dec. 2008.

[6] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

[7] C. Di Martino, Z. Kalbarczyk, R. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 610–621, June 2014.

[8] J. Domke, T. Hoefler, and S. Matsuoka. 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, pages 597–608, Piscataway, NJ, USA, 2014. IEEE Press.

[9] H. Fujita, N. Dun, A. Fang, Z. A. Rubenstein, Z. Zheng, K. Iskra, J. Hammond, A. Dubey, P. Balaji, and A. A. Chien. 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, 2014.

[10] IBM. *IBM System Blue Gene Solution: Blue Gene/Q Application Development*, Second Edition edition, 2013.

[11] IBM. *IBM System Blue Gene Solution: Blue Gene/Q System Administration*, Second Edition edition, 2013.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[13] A. J. Peña, R. G. C. Carvalho, J. Dinan, P. Balaji, R. Thakur, and W. Gropp. Analysis of Topology-dependent MPI Performance on Gemini Networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 61–66, New York, NY, USA, 2013. ACM.

[14] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. 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, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[15] N. Shida, S. Sumimoto, and A. Uno. MPI library and low-level communication on the K computer. *FUJITSU Scientific & Technical Journal*, 48(3):324–330, July 2012.

[16] M. Stephan. JUQUEEN: Blue Gene/Q - System Architecture, 2012. http://www.training.prace-ri.eu/uploads/tx_pracetmo/JUQUEENSystemArchitecture.pdf.

[17] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. TOP500 SUpermputer Site. http://www.top500.org/.

[18] S. Sumimoto. The MPI Communication Library for K computer: Its Design and Implementation. Invited talk at EuroMPI 2012 in Vienna.

[19] A. Takefusa, T. Ikegami, H. Nakada, R. Takano, T. Tozawa, and Y. Tanaka. 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, 2014.

[20] K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM.

[21] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu. Comparison Research Between XY and Odd-Even Routing Algorithm of a 2-Dimension 3X3 Mesh Topology Network-on-Chip. In *Proceedings of the 2009 WRI Global Congress on Intelligent Systems - Volume 03*, GCIS '09, pages 329–333, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Z. Zheng, A. Chien, and K. Teranishi. Fault Tolerance in an Inner-outer Solver: A GVR-enabled Case Study, 2014. http://www.vecpar.org/papers/vecpar2014_submission_4.pdf.