Task and Core-based Automated Fault Tolerance in High-Performance Computing Systems

Blesson Varghese, *Member, IEEE,* Gerard McKee, *Member, IEEE,* and Vassil Alexandrov, *Member, IEEE*

Abstract—High-performance computing systems require manual intervention if one or more computing cores fail. This places a cost on the maintenance of computing tasks. Intelligent approaches which can proactively detect computing core failures and take action to relocate the computing core's task onto reliable cores can make a significant step towards automating fault tolerance in high-performance computing systems. This paper describes an experimental investigation into multi-agent approaches to bring in this intelligence to high-performance computing systems. Three approaches are studied to incorporate agent intelligence with high-performance computing systems; the first at the task level, the second at the core level and the third both at the task and core level. The approaches are investigated for single core failure scenarios that can occur in the execution of parallel reduction algorithms on computer clusters. The key result is that a task can be relocated without manual intervention and with a time delay in the order of milliseconds.

Index Terms—Automated Fault Tolerance, Intelligent Agents, Intelligent Cores, Multi-Agent System.

1 Introduction

The scale of resources and computations in high-performance computing systems is significantly increasing. With this increase the resultant number of failures will increase and the time towards a failure will decrease [1][2][3][4]. A key challenge in maintaining the operation of high-performance computing systems in the event of failure is addressed under research in fault tolerance.

The conventional fault tolerant mechanism that is employed in many computing systems is checkpointing, which involves the periodic recording of intermediate states of execution of a process to which execution can be returned if a fault occurs. Such traditional fault tolerant mechanisms, however, are challenged by drawbacks such as single point failures, lack of scalability and communication overheads, which pose a constraint in achieving efficient fault tolerance when applied to high-performance computing systems.

- *B. Varghese is a Postdoctoral Fellow at the Faculty of Computer Science, Dalhousie University, Halifax, Canada. E-mail: varghese@cs.dal.ca; Webpage: http://www.blessonv.com (Corresponding Author)*
- *G. McKee is the Dean of the Faculty of Computing and IT, Baze University, Abuja, Nigeria. Formerly, he was Senior Lecturer in Networked Robotics at the School of Systems Engineering, University of Reading, UK. E-mail: gerard.mckee@bazeuniversity.edu.ng*
- *V. Alexandrov is ICREA Research Professor in Computational Science at the Barcelona Supercomputing Centre, Barcelona, Spain. Formerly, he was the Director of the Centre for Advanced Computing and Emerging Technologies (ACET) and Professor in Computational Science at the School of Systems Engineering, University of Reading, UK. E-mail:vassil.alexandrov@bsc.es*

Moreover, many of the traditional fault tolerant mechanisms are manual methods and require human administrator interventions to isolate recurring faults. Self-managing fault tolerant mechanisms are therefore required. However, such mechanisms are not readily available. Therefore, the objective of the research reported in this paper is the development of approaches which incorporate automated methods for fault tolerance.

Three approaches, firstly, an approach incorporating agent intelligence, secondly, an approach incorporating core intelligence, and thirdly, a hybrid approach are proposed as means of achieving both the computation and incorporating self-managing fault tolerance. In the first approach, automated fault tolerance is achieved by a collection of agents which can freely traverse on a network of computing cores. Each agent carries a task to be executed on a computing core in the form of a payload. Fault tolerance in this context can be achieved since an agent can move on the network of cores, which is effectively moving a task from one computing core onto another.

In the second approach, automated fault tolerance is achieved by considering the computing cores as a landscape to be intelligent. The cores can move processes executed on them across the landscape. Fault tolerance in this context can be achieved since a core can migrate a process executed on it onto another core.

In the third approach, these two forms of intelligence are combined in a hybrid approach. The task to be executed on a computing core is mapped onto a set of agents which are released onto the landscape, which is in turn an abstraction of the network of computing cores. The three approaches operate at the middle levels, transforming the set of agents into fault-tolerance-aware agents and the computational platform into fault-tolerance-aware cores, and a combination of these.

The remainder of this paper is organised as follows. Section 2 presents a review of work relevant to fault tolerance. The following three sections propose three fault tolerant approaches - Section 3 considers the first approach which incorporates agent-based intelligence, Section 4 considers the second approach which incorporates core-based intelligence and Section 5 considers a hybrid approach which combines the first and second approach. Experimental studies are presented in Section 6. The paper is concluded in Section 7.

2 Literature Review

A review of literature related to fault tolerance led to a classification of research in fault-tolerance. The classifications are as follows:

(i) Based on when a response to mitigate a failure is initiated relative to the occurrence of the failure.
(ii) Based on where the fault tolerant strategy is implemented.
(iii) Based on the study of faults for implementing fault tolerant strategies.
(iv) Based on the spatial locality of the faults.
(v) Based on when a fault tolerant strategy is selected or when it comes to play.
(vi) Based on the location of control of the fault tolerant strategies.
(vii) Based on the dependence on the algorithm being executed.
(viii) Based on the underlying strategy.

Fault tolerance based on when a response to mitigate a failure is initiated relative to the occurrence of the failure can be separated out as proactive, reactive and adaptive. In proactive fault tolerance, an attempt is made to predict a failure of a compute resource before it occurs and then relocate a task executing on it onto another resource. For example, proactive fault tolerance can be achieved by failure prediction followed by process migration presented in [5], [6], [7], [8].

In reactive fault tolerance, an attempt is made to minimise the impact of a failure after it has occurred. Traditional mechanisms such as checkpointing [9], [10], rollback recovery [11], [12], [13], [14] and message logging [15], [16], [17] are useful for reactive fault tolerance.

A hybrid of both reactive and proactive fault tolerance, referred to as adaptive fault tolerance, is implemented so that failures that cannot be predicted in proactive strategies are handled by the reactive strategies [18], [19], [20], [21], [22], [23].

Based on which layer of a system the fault tolerant strategies are implemented can be separated out as hardware, middleware, application (even called as software or algorithm) fault tolerance. Hardware fault tolerance considers low-level failure mitigation of the processor or its circuitry [24], [25], [26]. In this level, transient faults commonly occur which changes the state of a transistor [27]. Strategies to overcome failures are often derived through physical fault injection [28], [29] through (a) the processor pins [30], (b) heavy ion radiation [31], (c) electromagnetic interference [32] and (d) lasers [33].

Middleware fault tolerant strategies are incorporated between an application and the underlying software. A programmer is provided with the flexibility to incorporate methods, procedures and functions whose fault tolerance is dealt by the supporting libraries. For example, FT-MPI (Fault Tolerant Message Passing Interface) is an extension of the MPI library incorporating fault tolerant functions [34]. Checkpointing is incorporated in a number of middleware libraries such as [35] and [36].

Application based fault tolerant strategies are implemented within algorithms that are not fault tolerant inherently. A programmer can (a) include a strategy that is supported by the middleware and embed it within the algorithm [37], or (b) incorporate a strategy that may be invoked during execution or (c) invoked at run-time [38]. Failsafe fault tolerance [39], [40] and Algorithm-based fault tolerance (ABFT) [41], [42], [43], [44] are two examples. Other application fault tolerant strategies are reported in [45], [46], [47], [48], [49], [50], [51].

Offline strategies and Production/Online strategies are two distinctions based on the study of faults for implementing fault tolerant strategies. Offline methods are typically used to remove design and implementation faults and to derive fault tolerant strategies. Fault injection is an example of offline methods in which faults are generated using hardware-based (or physical considered above), software-based, simulation-based, emulation-based or a hybrid of one or more models [52], [29]. A wide variety of tools that incorporate these models are available. For example, FERRARI [53], LFI [54] and Xception [55]. Datamining is another technique for offline strategies [56].

Production/online strategies are incorporated for real-time use. Checkpointing, a conventional strategy is a typical example [9], [10], [57].

The spatial locality of where failures occur determines the strategies that need to be chosen for fault tolerance. Local failures ranging due to a single node to a small collection of nodes or related software component(s), may only require conventional centralised (single or multiple server) checkpointing [58]. However, non-local failures that are dispersed geographically for a large-scale system may require a scalable [59], [60], [61], [62], distributed [9] and diskless checkpointing strategy [63], [64], [65].

The control of a fault tolerant strategy can either be centralised or distributed. In strategies incorporating centralised fault tolerant control, there may be a single server used for backup and a single daemon responsible for monitoring processes that are executed on a network of nodes. For example, a traditional and centralised message logging or checkpoint strategy is a classic example. Such mechanisms (a) are susceptible to single point of failures, (b) cannot scale over a large network of nodes, (c) have large overheads, and (d) require large disk storage.

Distributed fault tolerant control on the other hand lends for greater scalability and reliability. Distributed diagnosis [66], [67], distributed checkpointing [9], [68], [69] and diskless checkpointing [70], [63], [64] are strategies that address distributed fault tolerant control.

Fault tolerance based on the dependence of algorithms can be separated out as algorithm dependent and algorithm independent. Algorithm dependent strategies, for example as presented in [71], are strategies that may be computationally effective for a specific system. Algorithm independent strategies may be applied to any application. For example, middleware strategies are algorithm independent.

Based on the underlying mechanism incorporated, fault tolerant strategies can be identified as checkpoint-based, replication-based or redundancy-based, migration-based and agent-based. Checkpoint-based mechanisms can either be further separated out on the basis of strategies [72] and coordination [78]. MPI checkpointing [10], [9], optimal checkpointing [73], [74], cooperative checkpointing [75], [76] and adaptive checkpointing [72], [77] are based on strategies. Based on coordination, checkpointing can be separated as uncoordinated checkpointing [79], [80], coordinated checkpointing [81], [82] and communication induced checkpointing [83], [84].

Replication-based mechanisms can replicate either one or a set of (a) the program, (b) the executing process, (c) the data employed by the process, (d) the state of the process and (d) the checkpoints. Replication-based mechanisms are reported in [85], [1], [58], [86].

Migration-based mechanisms [87], [6], [88], [89] are important to relocate a process from one computing node onto another. Migration can be (a) based on state information [90] - minimal state, full state or distributed state, (b) based on level of migration [91] - kernel-level, user-level or application-level, (c) based on hierarchical level - high level [92] or low-level [93] and (d) based on geographic nature - geographically dispersed [94] or geographically closed [95].

Agent-based mechanisms support mobility and coordination that are useful for distributed fault tolerance strategies. In [96], the fault tolerant strategy incorporates three types of agents, namely the actual agent, the witness agent and the probe agent for exchanging data and resource sharing. Autonomous Cooperation System (ACS) [97], FAult TOlerance of the Mobile Agent System (FATOMAS) [98], Dynamic Agent Replication eXtension (DARX) [99], the Fault Tolerant Control System in [100] and DimaX [101] are examples.

In the next sections, three fault tolerant approaches for high-performance computing systems are proposed and implemented. These approaches are: (i) based on proactive fault tolerance, (ii) implemented on an algorithm level, (iii) available for production, (iv) capable to handle non-local faults, (v) controlled in a distributed manner, (vi) algorithm independent, and (vii) migration and agent based strategies. The first approach proposed in Section 3 considers agent intelligence which can be achieved through process migration. The second approach presented in Section 4 incorporates core intelligence using processor virtualisation. A hybrid combining both forms of intelligence is incorporated in the third approach considered in Section 5.

3 Approach 1: Fault Tolerance Incorporating Agent Intelligence

A task, T, which needs to be executed on a large-scale system is decomposed into a set of sub-tasks $T_1, T_2 \cdots T_n$. Each sub-task $T_1, T_2 \cdots T_n$ is mapped onto agents $A_1, A_2 \cdots A_n$ that carry the sub-tasks as payloads onto the cores, $C_1, C_2 \cdots C_n$ of the landscape as shown in Figure 1. The agents and the sub-task are independent of each other; in other words, an agent acts as a wrapper around a sub-task to situate the sub-task on a core.

There are three computational requirements of the agent to achieve successful execution of the task: (a) The agent needs to know the overall task, T, that needs to be achieved, (b) The agent needs to access

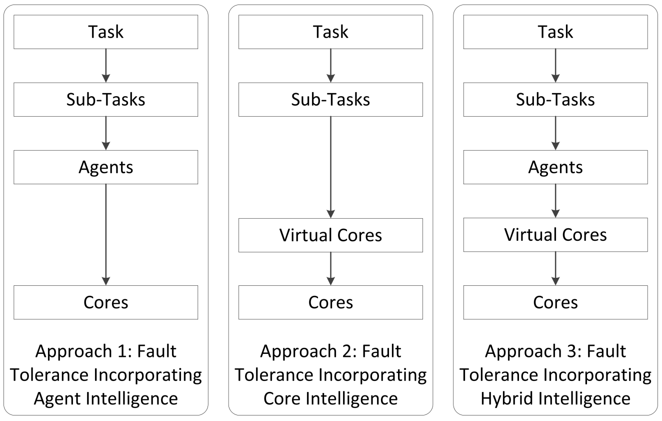

Task
Sub-Tasks
Agents
Cores
Approach 1: Fault Tolerance Incorporating Agent Intelligence
Task
Sub-Tasks
Virtual Cores
Cores
Approach 2: Fault Tolerance Incorporating Core Intelligence
Task
Sub-Tasks
Agents
Virtual Cores
Cores
Approach 3: Fault Tolerance Incorporating Hybrid Intelligence

Fig. 1: The task, sub-tasks, agents, virtual cores and computing cores in the three approaches proposed for automated fault tolerance

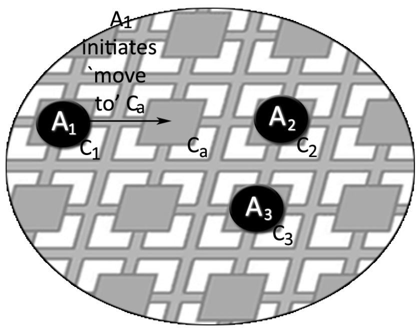


Fig. 2: Agents A_1, A_2 and A_3 are situated on cores C_1, C_2 and C_3 respectively. A failure is predicted on core C_1. The agent A_1 moves onto core C_a.

data required by the sub-task it is carrying and (c) The agent needs to know the operation that the sub-task needs to perform on the data. The agents then displace across the landscape to compute the sub-tasks.

Intelligence of an agent can be useful in at least four important ways for achieving fault tolerance while a sub-task is executed. Firstly, an agent knows the landscape in which it is located. Knowledge of the landscape is threefold which includes (a) the knowledge of the computing core on which the agent is located, (b) knowledge of other computing cores in the vicinity of the agent and (c) knowledge of agents located in the vicinity.

Secondly, an agent identifies a location to situate within the landscape. This is possible by gathering information from the vicinity using probing processes and is required when the computing core on which the agent is located is anticipated to fail.

Thirdly, an agent predicts failures that are likely to impair its functioning. The prediction of failures (for example, due to the failure of the computing core) is along similar lines to proactive fault tolerance.

Fourthly, an agent is mobile within the landscape. If the agent predicts a failure then the agent can relocate onto another computing core thereby moving off the task from the core anticipated to fail (refer Figure 2).

The intelligence of agents is incorporated within the following sequence of steps that describes an approach for fault tolerance:

Agent Intelligence Based Fault Tolerance

```
Step 1: Decompose a task, $T$, to be executed on the
        landscape into sub-tasks, $T_1, T_2 \cdots T_n$
Step 2: Each sub-task provided as a payload to
        agents, $A_1, A_2 \cdots A_n$
Step 3: Agents carry tasks onto computing cores,
        $C_1, C_2 \cdots C_n$
Step 4: For each agent, $A_i$ located on computing
        core $C_i$, where $i = 1$ to $n$
    Step 4.1: Periodically probe the computing
              core $C_i$
    Step 4.2: if $C_i$ predicted to fail, then
        Step 4.2.1: Agent, $A_i$ moves onto
                    an adjacent computing
                    core, $C_a$
        Step 4.2.2: Notify dependent
                    agents
        Step 4.2.3: Agent $A_i$ establishes
                    dependencies
Step 5: Collate execution results from sub-tasks
```

3.1 Failure Scenarios

Two core failure scenarios are considered for the agent intelligence based fault tolerance concept. In the first scenario, it is assumed that there are no failures of cores adjacent to the core anticipated to fail in any time step of the execution of a task. In the second scenario, however, a failure of a core adjacent to the core predicted to fail is possible.

The communication sequence in the first scenario as shown in Figure 3 is as follows. The hardware probing process on the core anticipating failure, C_{PF} notifies the failure prediction to the agent process, P_{PF}, situated on it. The agent then creates a new process on an adjacent core and transfers data it was using onto the newly created process. Then the input dependent ($P_{ID1} \cdots P_{IDn}$) and output dependent ($P_{OD1} \cdots P_{ODn}$) processes are notified. The agent process on C_{PF} is terminated thereafter. The new agent process on the adjacent core establishes dependencies with the input and output dependent processes.

The communication sequence in the second scenario as shown in Figure 4 is similar to the first scenario. The hardware probing process on the core anticipating failure, C_{PF} notifies the failure prediction to the agent process, P_{PF}, situated on it. However, in the second scenario since the failure of a core adjacent to the core predicted to fail is possible it is necessary that the predictions of the hardware probing processes on the adjacent cores be requested. Once the predictions are gathered, the agent process, P_{PF}, creates a new process on an adjacent core and transfers data to it. The input and output dependent processes are notified and the agent dependencies with these processes are reinstated. The agent process, P_{PF} is terminated.

4 Approach 2: Fault Tolerance Incorporating Core Intelligence

A task, T, which needs to be executed on a large-scale system is decomposed into a set of sub-tasks $T_1, T_2 \cdots T_n$. Each sub-task $T_1, T_2 \cdots T_n$ is mapped onto the virtual cores, $VC_1, VC_2 \cdots VC_n$, an abstraction over $C_1, C_2 \cdots C_n$ respectively as shown in Figure 1. The cores referred to in this approach are virtual cores which are an abstraction over the hardware

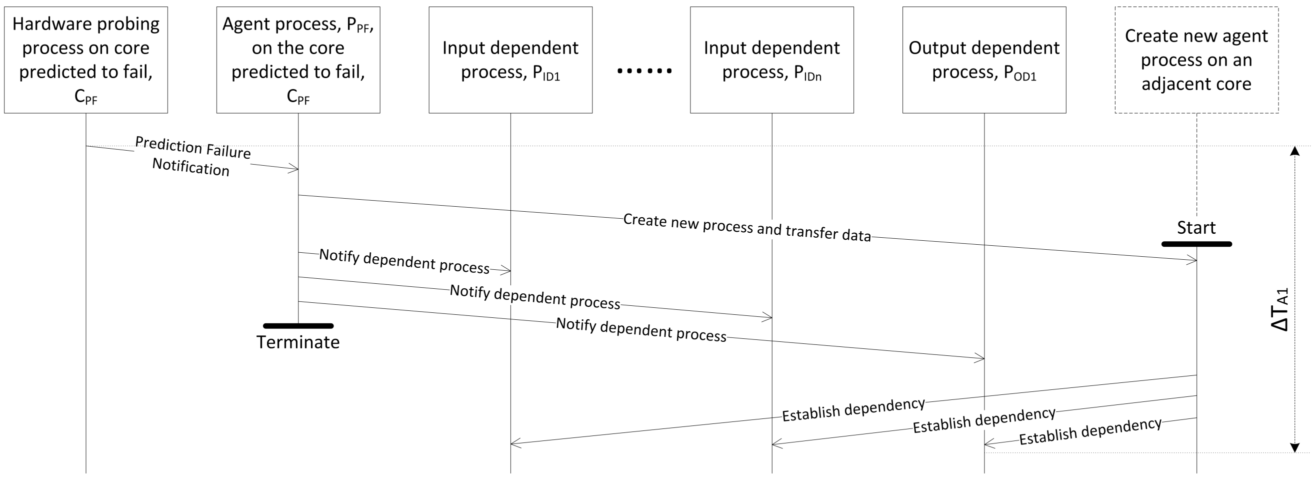

Hardware probing process on core predicted to fail, C_PF
Agent process, P_PF, on the core predicted to fail, C_PF
Input dependent process, P_ID1
Input dependent process, P_IDn
Output dependent process, P_OD1
Create new agent process on an adjacent core
Prediction Failure Notification
Create new process and transfer data
Start
Notify dependent process
Notify dependent process
Notify dependent process
Terminate
Establish dependency
Establish dependency
Establish dependency
ΔT_A1

Fig. 3: Communication sequence in the first scenario of agent intelligence based fault tolerance

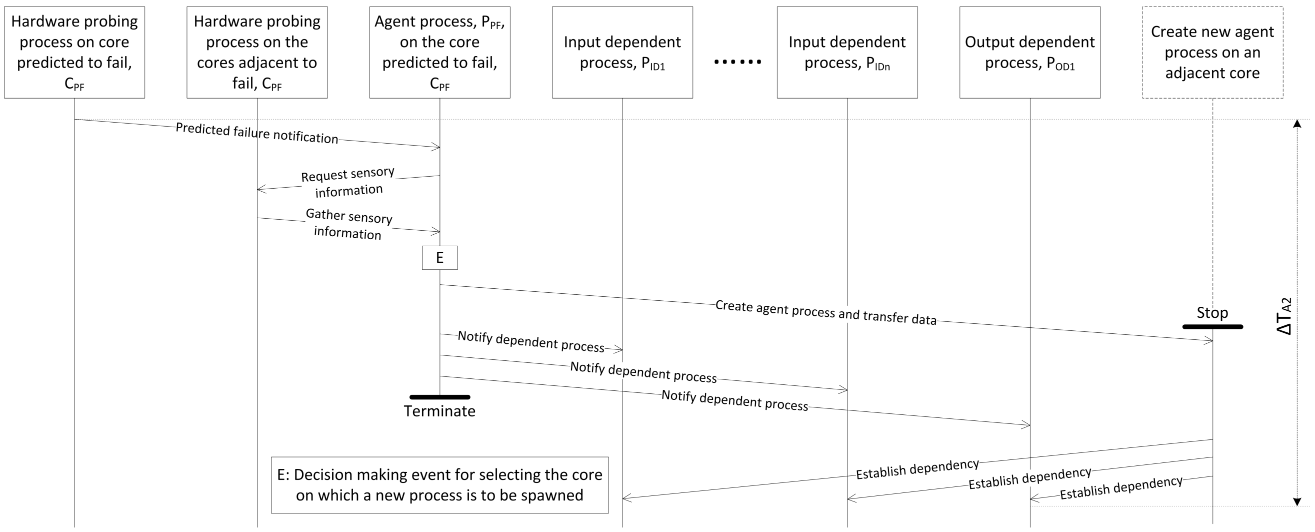

Hardware probing process on core predicted to fail, C_PF
Hardware probing process on the cores adjacent to fail, C_PF
Agent process, P_PF, on the core predicted to fail, C_PF
Input dependent process, P_ID1
Input dependent process, P_IDn
Output dependent process, P_OD1
Create new agent process on an adjacent core
Predicted failure notification
Request sensory information
Gather sensory information
E
Create agent process and transfer data
Stop
Notify dependent process
Notify dependent process
Notify dependent process
Terminate
E: Decision making event for selecting the core on which a new process is to be spawned
Establish dependency
Establish dependency
Establish dependency
ΔT_A2

Fig. 4: Communication sequence in the second scenario of agent intelligence based fault tolerance

computing cores. The virtual cores are a logical representation and may incorporate rules to achieve intelligent behaviour.

Intelligence of a core is useful in a number of ways for achieving fault tolerance. Firstly, a core updates knowledge of its surrounding by monitoring adjacent neighbours. Independent of what the cores are executing, the cores can monitor each other. Each core can ask the question 'are you alive?' to its neighbours and gain information.

Secondly, a core periodically updates information of its surrounding. This is useful for the core to know which neighbouring cores can execute a task if it fails.

Thirdly, a core periodically monitors itself using a hardware probing process and predicts if a failure is likely to occur on it.

Fourthly, a core can move a task executing on it onto an adjacent core if a failure is expected and adjust to failure as shown in Figure 5. Once a task has relocated all data dependencies will need to be re-established.

The following sequence of steps describe an approach for fault tolerance incorporating core intelligence:

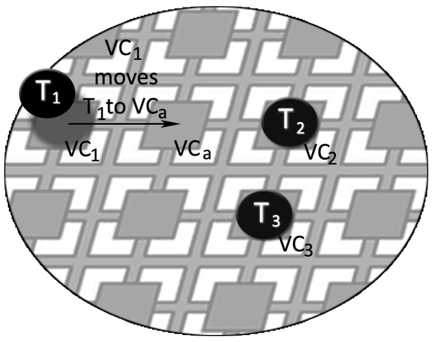


Fig. 5: Tasks T_1, T_2 and T_3 are situated on virtual cores VC_1, VC_2 and VC_3 respectively. A failure is predicted on core C_1 and VC_1 moves the task T_1 onto virtual core VC_a.

Core Intelligence Based Fault Tolerance

Step 1: Decompose a task, T, to be executed on the

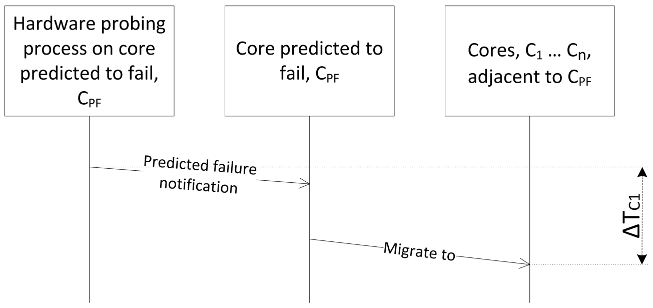

Hardware probing process on core predicted to fail, C_PF
Core predicted to fail, C_PF
Cores, C_1 … C_n, adjacent to C_PF
Predicted failure notification
Migrate to
ΔT_C1

Fig. 6: Communication sequence in the first scenario of core intelligence based fault tolerance

```
          landscape into sub-tasks, $T_1, T_2 \cdots T_n$
Step 2: Each sub-task allocated to cores, $VC_1, VC_2 \cdots VC_n$
Step 3: For each core, $VC_i$, where $i = 1$ to $n$ until sub-task $T_i$ completes execution
    Step 3.1: Periodically probe the computing core $C_i$
    Step 3.2: if $C_i$ predicted to fail, then
        Step 3.2.1: Migrate sub-task $T_i$ on $VC_i$ onto an adjacent computing core, $VC_a$
Step 4: Collate execution results from sub-tasks
```

4.1 Failure Scenarios

Two core failure scenarios are considered for the agent intelligence based fault tolerance concept. The first and the second scenarios are both similar to the failure scenarios considered for the approach incorporating agent intelligence.

In the first scenario as shown in Figure 6, the communication sequence is as follows. The hardware probing process on the core predicted to fail, C_{PF} notifies a predicted failure to the core. The task executed on VC_{PF} is migrated onto an adjacent core $VC_1 \cdots VC_n$.

The second scenario as shown in Figure 7, is similar to the first scenario. The predictions of the hardware probing processes on the adjacent cores are requested once the core predicted to fail is notified of the failure. The task executed on VC_{PF} is then migrated onto an adjacent core once a decision based on failure predictions are received from the hardware probing processes of adjacent cores.

5 Approach 3: Fault Tolerance Incorporating Hybrid Intelligence

The hybrid approach acts as an umbrella bringing together the concepts of agent intelligence and core intelligence considered in Section 3 and Section 4. The key concept of the hybrid approach lies is the mobility of the agents on the cores and the cores collectively executing a task. Decision-making is required in this

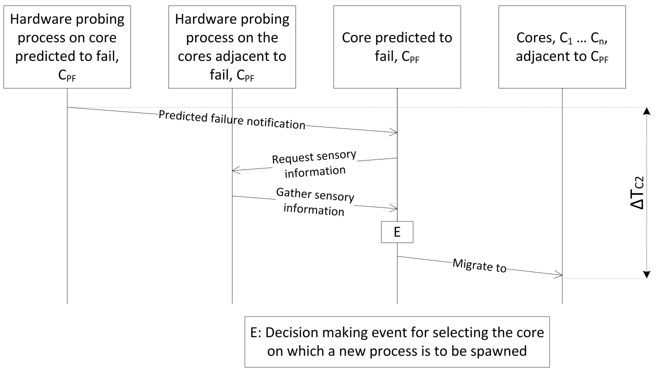

Hardware probing process on core predicted to fail, C_PF
Hardware probing process on the cores adjacent to fail, C_PF
Core predicted to fail, C_PF
Cores, C_1 … C_n, adjacent to C_PF
Predicted failure notification
Request sensory information
Gather sensory information
E
Migrate to
ΔT_C2
E: Decision making event for selecting the core on which a new process is to be spawned

Fig. 7: Communication sequence in the second scenario of core intelligence based fault tolerance

approach for choosing between the agent intelligence and core intelligence approaches when a failure is expected.

The hybrid intelligence is incorporated within the following sequence of steps that describes an approach for fault tolerance:

Hybrid Intelligence Based Fault Tolerance

```
Step 1: Decompose a task, $T$, to be executed on the landscape into sub-tasks, $T_1, T_2 \cdots T_n$
Step 2: Each sub-task provided as a payload to agents, $A_1, A_2 \cdots A_n$
Step 3: Agents carry tasks onto virtual cores, $VC_1, VC_2 \cdots VC_n$
Step 4: For each agent, $A_i$ located on virtual core $VC_i$, where $i = 1$ to $n$
    Step 4.1: Periodically probe the computing core $C_i$
    Step 4.2: if $C_i$ predicted to fail, then
        Step 4.2.1: if 'Agent Intelligence' is a suitable mechanism, then
            Step 4.2.1.1: Agent, $A_i$, moves onto an adjacent computing core, $VC_a$
            Step 4.2.1.2: Notify dependent agents
            Step 4.2.1.3: Agent $A_i$ establishes dependencies
        Step 4.2.2: else if 'Core Intelligence' is a suitable mechanism, then
            Step 4.2.2.1: Core $VC_i$ migrates agent, $A_i$ onto an adjacent computing core, $VC_a$
Step 5: Collate execution results from sub-tasks
```

When a core failure is anticipated both an agent and a core can make decisions which can lead to a conflict. For example, an agent can attempt to move onto an adjacent core while a core would like to migrate the agent onto an adjacent core. Therefore, an agent and

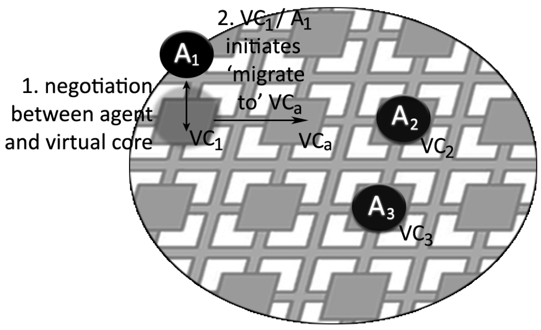

2. VC1/ A1 initiates 'migrate to' VCa
1. negotiation between agent and virtual core
A1
VC1
VCa
A2
VC2
A3
VC3

Fig. 8: Avoiding a conflict in decision-making between an agent and a core. Agents A_1, A_2 and A_3 are situated on virtual cores VC_1, VC_2 and VC_3 which are mapped onto computing cores C_1, C_2 and C_3 respectively. A failure is predicted on core C_1. The agent A_1 and VC_1 negotiate to decide who moves the sub-task onto core VC_a.

the core on which it is located need to negotiate before either of them initiate a response to move (see Figure 8). The negotiations need to be quick, and therefore, the rules for negotiation need to be established. Rules for decision making to avoid a conflict between an agent's and a core's decision will be considered in Section 6.2.3.

6 Experimental Studies

The implementation of the fault tolerant approaches is reported in this section. The platform and the results of the experiments on the agent and core approaches in the first and second failure scenario are considered.

6.1 Experimental Platform

Four computer clusters were used for the experimental studies reported in this section. The first was a cluster available at the Centre for Advanced Computing and Emerging Technologies (ACET), University of Reading, UK. Thirty three compute nodes connected through Gigabit Ethernet were available, each with Pentium IV processors and 512MB-2GB RAM.

The remaining three clusters are compute resources, namely Brasdor, Glooscap and Placentia, all provided by The Atlantic Computational Excellence Network (ACEnet) [102], Canada. Brasdor comprises 306 compute nodes connected through Gigabit Ethernet, with 932 cores and 1-2GB RAM. Glosscap comprises 97 nodes connected through Infiniband, with 852 cores and 1-8GB RAM. Placentia comprises 338 compute nodes connected through Infiniband, with 3740 cores and 2-16GB RAM

The cluster implementations in this paper are based on the Message Passing Interface (MPI). The first approach incorporating agent intelligence is implemented using Open MPI [103], an open source implementation of MPI 2.0. The dynamic process model which supports dynamic process creation and management facilitates control over an executing process. This feature is useful for implementing the first approach. The MPI functions useful in the implementation of the first approach are (i) MPI_COMM_SPAWN which creates a new MPI process and establishes communication with an existing MPI application and (ii) MPI_COMM_ACCEPT and MPI_COMM_CONNECT which establishes communication between two independent processes.

The second approach incorporating core intelligence is implemented using Adaptive MPI (AMPI) [104], developed over Charm++ [105], a C++ based parallel programming language. The aim of AMPI is to achieve dynamic load balancing by migrating objects over virtual cores and thereby facilitates control over cores. Core intelligence harnesses this potential of AMPI to migrate a task from a core onto another core. A strategy to migrate a task using the concepts of processor virtualization and dynamic task migration in AMPI and Charm++ is reported in [8].

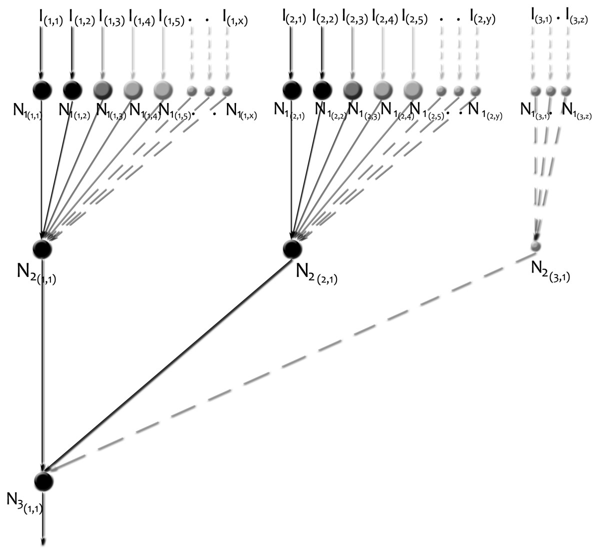
Parallel reduction algorithms which implement the bottom-up approach (i.e., data flows from the leaves to the root) are employed for the experiments. These algorithms are of interest for three reasons. Firstly, the algorithm lends itself to be easily decomposed into a set of sub-tasks. Each sub-task can then be mapped onto a computing core either by providing the sub-task as a payload to an agent in the first approach or by providing the task onto a virtual core incorporating intelligent rules.

Secondly, the execution of a parallel reduction algorithm stalls and produces incorrect solutions if a core fails. Therefore, parallel reduction algorithms can benefit from fault-tolerant techniques that can be incorporated within them.

Thirdly, parallel reduction algorithms are often employed in a number of domains. Incorporating self-managing fault tolerant mechanisms can make these algorithm more robust and reliable for applications which do not have the luxury of time for reinstating [106].

Figure 9 is an illustration of a generic parallel summation algorithm with three sets of input. Firstly, $I_{(1,1)}$, $I_{(1,2)}$ $\cdots$ $I_{(1,x)}$, secondly, $I_{(2,1)}$, $I_{(2,2)}$ $\cdots$ $I_{(2,y)}$, and thirdly, $I_{(3,1)}$ $\cdots$ $I_{(3,z)}$. The first level nodes which receive the three sets of input comprise three set of nodes. Firstly, $N_{1(1,1)}$, $N_{1(1,2)}$ $\cdots$ $N_{1(1,x)}$, secondly, $N_{1(2,1)}$, $N_{1(2,2)}$ $\cdots$ $N_{1(2,y)}$, and thirdly, $N_{1(3,1)}$, $N_{1(3,2)}$ $\cdots$ $N_{1(3,z)}$. The next level of nodes, $N_{2(1,1)}$, $N_{2(2,1)}$ and $N_{3(3,1)}$ receive inputs from the first level nodes. The resultant from the second level nodes is fed in to the third level node $N_{3(1,1)}$. Parallel summation is an exemplar of parallel reduction algorithms in which the nodes reduce the input through the output using the $\oplus$ operator.

The parallel summation algorithm can benefit from the inclusion of fault tolerant strategies. The task,



$I_{(1,1)}$ $I_{(1,2)}$ $I_{(1,3)}$ $I_{(1,4)}$ $I_{(1,5)}$ $I_{(1,x)}$
$I_{(2,1)}$ $I_{(2,2)}$ $I_{(2,3)}$ $I_{(2,4)}$ $I_{(2,5)}$ $I_{(2,y)}$
$I_{(3,1)}$ $I_{(3,z)}$
$N_{1(1,1)}$ $N_{1(1,2)}$ $N_{1(1,3)}$ $N_{1(1,4)}$ $N_{1(1,5)}$ $N_{1(1,x)}$
$N_{1(2,1)}$ $N_{1(2,2)}$ $N_{1(2,3)}$ $N_{1(2,4)}$ $N_{1(2,5)}$ $N_{1(2,y)}$
$N_{1(3,1)}$ $N_{1(3,z)}$
$N_{2(1,1)}$ $N_{2(2,1)}$ $N_{2(3,1)}$
$N_{3(1,1)}$

Fig. 9: Generic parallel summation algorithm

T, in this case is summation, and the sub-tasks, $T_1, T_2 \cdots T_n$ is also summation. In the first fault tolerant approach incorporating agent intelligence, the data to be summed along with the summation operator is provided to the agent. The agents locate on the computing cores and continuously probe the core for anticipating failures. If an agent is notified of a failure, then it moves off onto another computing core in the vicinity, thereby not stalling the execution towards achieving the summation task.

In the second fault tolerant approach incorporating core intelligence, the sub-task comprising the data to be summed along with the summation operator is located on the a virtual core. When the core anticipates a failure, it migrates the sub-task onto another core.

6.2 Experimental Results

Figures 10 and 11 are collections of graphs plotted using the parallel summation algorithm as a case study for both the first and second fault tolerant mechanisms. Each graph comprises four plots, the first representing the ACET cluster and the other three representing the three ACEnet clusters. Figure 10 shows graphs plotted for the first failure scenario, and Figure 11 shows graphs plotted for the second failure scenario. Both failure scenarios take into account the following three factors that can affect performance of the approaches:

(i) The number of dependencies of the subtask being executed denoted as Z. If the total number of input dependencies is d_i and the total number of output dependencies is d_o, then $Z = d_i + d_o$. For example, in a parallel summation algorithm incorporating binary trees, each node has two input dependencies and one output dependency, and therefore $Z = 3$. In the experiments, the number of dependencies is varied between 3 and 63, by changing the number of input dependencies of an agent or a core.

(ii) The size of the data communicated across the cores denoted as S_d. In the experiments, the input data is a matrix for parallel summation and its size is varied between 2^{19} to 2^{31} KB.

(iii) The process size of the distributed components of the task denoted as S_p. In the experiments, the process size is varied between 2^{19} to 2^{31} KB which is proportional to the input data.

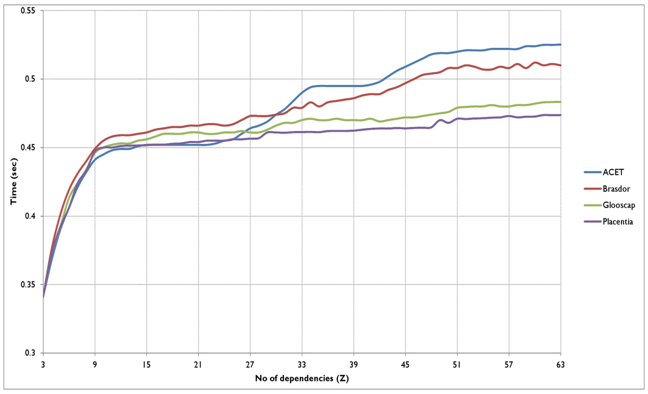
6.2.1 *First Failure Scenario*

Figure 10a is a graph of the time taken in seconds for reinstating execution versus the number of dependencies in the first failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A1}, is computed for varying numbers of dependencies, Z ranging from 3 to 63. The size of the data on the agent is $S_d = 2^{24}$ kilo bytes. The approach is slowest on the ACET cluster and fastest on the Placentia cluster. In all cases the communication overheads result in a steep rise in the time taken for execution until $Z = 10$. The time taken on the ACET cluster rises once again after $Z = 25$.

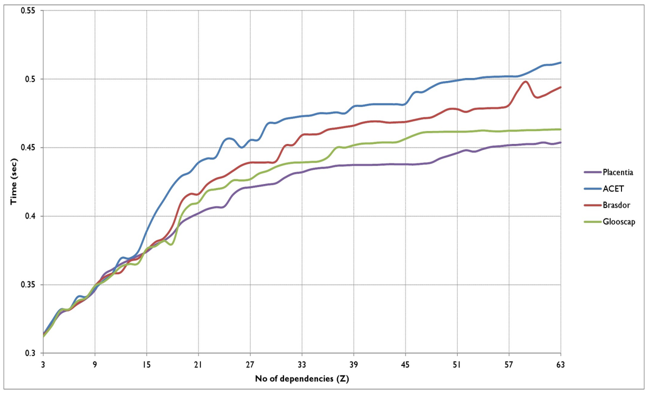
Figure 10b is a graph of the time taken in seconds for reinstating execution versus the number of dependencies in the first failure scenario based on core intelligence. The mean time taken to reinstate execution, for 30 trials, ΔT_{C1}, is computed for varying number of dependencies, Z ranging from 3 to 63. The size of the data on the core is $S_d = 2^{24}$ kilo bytes. The approach requires almost the same time on the four clusters for reinstating execution until $Z = 10$, after which there is divergence in the plots. The approach is most effective on the Placentia cluster.

Figure 10c is a graph showing the time taken in seconds for reinstating execution versus the size of data in kilobytes (KB), $S_d = 2^n$, where $n = 19, 19.5 \cdots 31$, carried by an agent in the first failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A1}, is computed for varying sizes of data ranging from 2^{19} to 2^{31} KB. The number of dependencies Z is 10 for the graph plotted. The approach results in similar trends on the four clusters in which there is a gradual increase in the time taken as S_d increases. The approach exhibits the fastest performance on the Glooscap and Placentia clusters.

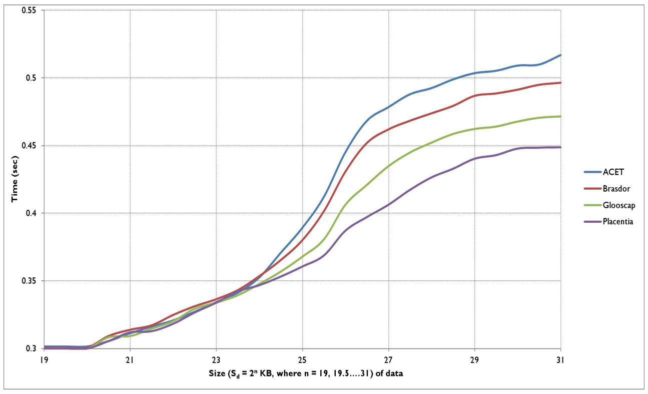
Figure 10d is a graph showing the time taken in seconds for reinstating execution versus the size of data in kilobytes (KB), $S_d = 2^n$), where $n = 19, 19.5 \cdots 31$, on a core in the first failure scenario based on core intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{C1}, is computed for varying sizes of data ranging from 2^{19} to 2^{31} KB. The approach has similar times on the four clusters until $n = 24$. The approach performs well on the Glooscap and Placentia clusters.



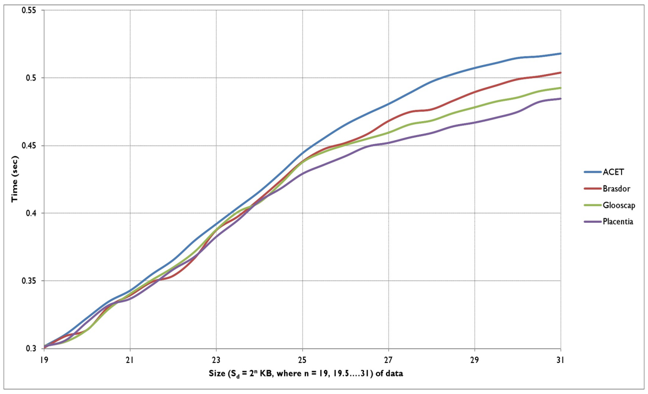
(a) No. of dependencies vs time taken for reinstating execution after failure in the agent intelligent approach



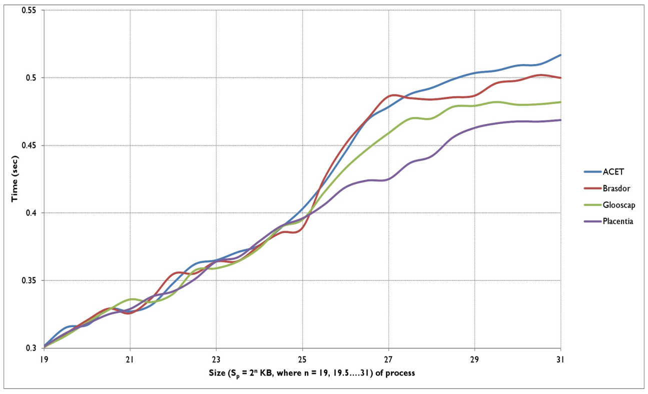
(b) No. of dependencies vs time taken for reinstating execution after failure in the core intelligent approach



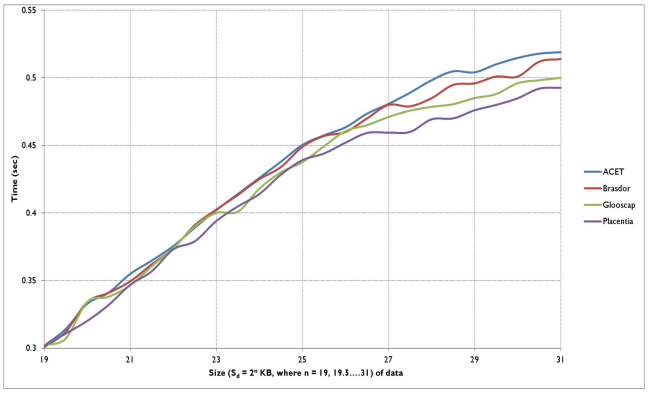
(c) Size of data vs time taken for reinstating execution after failure in the agent intelligent approach



(d) Size of data vs time taken for reinstating execution after failure in the core intelligent approach



(e) Process size vs time taken for reinstating execution after failure in the agent intelligent approach



(f) Process size vs time taken for reinstating execution after failure in the core intelligent approach

Fig. 10: Graphs plotted for the fault tolerant approaches in the first failure scenario

Figure 10e is a graph showing the time taken in seconds for reinstating execution versus the process size in kilobytes (KB), $S_p = 2^n$), where $n = 19, 19.5 \cdots 31$, in the first failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A1}, is computed for process sizes ranging from 2^{19} to 2^{31} KB. The approach shows interesting behaviour on the Brasdor cluster with a steep rise in the time taken between 2^{25} to 2^{27} KB. This approach performs well on the Glooscap and Placentia compared to ACET and Brasdor.

Figure 10f is a graph showing the time taken in seconds for reinstating execution versus the process size in kilobytes (KB), $S_p = 2^n$), where $n = 19, 19.5 \cdots 31$, in the first failure scenario based on core intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{C1}, is computed for process sizes ranging from 2^{19} to 2^{31} KB. In this graph all cluster show almost similar performance until 2^{26} KB. ACET and Brasdor have slightly higher times for large process sizes.

6.2.2 Second Failure Scenario

Figure 11a is a graph of the time taken in seconds for reinstating execution versus the number of dependencies in the second failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A2}, is computed for varying numbers of dependencies, Z ranging from 3 to 63. The size of the data on the agent is $S_d = 2^{24}$ kilo bytes. The results are similar to those obtained in the first scenario.

Figure 11b is a graph of the time taken in seconds for reinstating execution versus the number of dependencies in the second failure scenario based on core intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{C2}, is computed for varying number of dependencies, Z ranging from 3 to 63. The size of the data on the core is $S_d = 2^{24}$ kilo bytes. The results are similar to the first scenario and it is observed that the approach lends itself well on Placentia and Glooscap.

Figure 11c is a graph showing the time taken in seconds for reinstating execution versus the size of data in kilobytes (KB), $S_d = 2^n$, where $n = 19, 19.5 \cdots 31$, carried by an agent in the second failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A2}, is computed for varying sizes of data ranging from 2^{19} to 2^{31} KB. The number of dependencies Z is 10 for the graph plotted. Placentia and Glooscap outperforms ACET and Brasdor in the agent approach for varying size of data.

Figure 11d is a graph showing the time taken in seconds for reinstating execution versus the size of data in kilobytes (KB), $S_d = 2^n$, where $n = 19, 19.5 \cdots 31$, on a core in the second failure scenario based on core intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{C2}, is computed for varying sizes of data ranging from 2^{19} to 2^{31} KB. The number of dependencies Z is 10 for the graph plotted. In this graph, nearly similar time is taken by the approach on the four clusters with the ACET cluster requiring more time than the other clusters for $n > 24$.

Figure 11e is a graph showing the time taken in seconds for reinstating execution versus process size in kilobytes (KB), $S_p = 2^n$, where $n = 19, 19.5 \cdots 31$, in the second failure scenario based on agent intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{A2}, is computed for varying process sizes ranging from 2^{19} to 2^{31} KB. The number of dependencies Z is 10 for the graph plotted. The second scenario performs similar to the first scenario. The approach takes almost similar times to reinstate execution after a failure on the four clusters, but there is a diverging behaviour after $n > 26$.

Figure 11f is a graph showing the time taken in seconds for reinstating execution versus process size in kilobytes (KB), $S_p = 2^n$, where $n = 19, 19.5 \cdots 31$, in the second failure scenario based on core intelligence. The mean time taken to reinstate execution for 30 trials, ΔT_{C2}, is computed for varying process sizes ranging from 2^{19} to 2^{31} KB. The number of dependencies Z is 10 for the graph plotted. The approach has similar performance on the four clusters, though Placentia performs better than the other three clusters for more than 2^{26} KB.

6.2.3 Summary

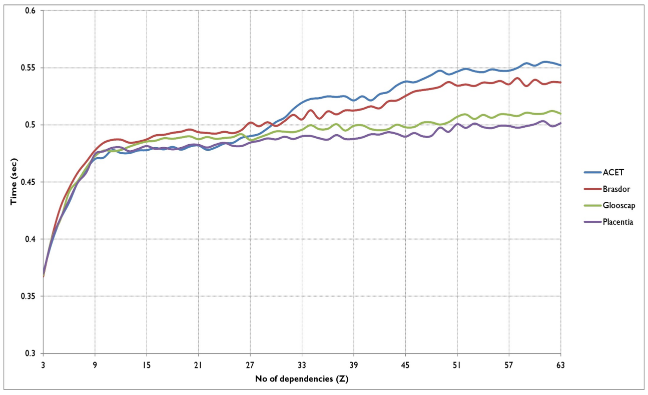
In the second failure scenario, a sub-task is moved off onto a core after the agent or the virtual core on the core anticipated to fail gathers information on whether its neighbouring cores are likely to fail. The agent or the virtual core make a decision based on this information which places a communication overhead when compared to the first scenario. The additional communication overhead in the second scenario is denoted as $\Delta T_o = \Delta T_{C2} - \Delta T_{C1}$ and $\Delta T_o = \Delta T_{A2} - \Delta T_{A1}$, which is the difference in the time taken for reinstating between the second and first failure scenario. The average communication overhead is computed as 0.028 seconds.

The number of dependencies, size of data, process size and communication overheads are the four factors taken into account in the experimental results. The results indicate that the approach incorporating core intelligence in both scenarios take lesser time than the approach incorporating agent intelligence. This is due to a two fold reason. Firstly, in the agent approach, the agent needs to establish the dependency with each agent individually, where as in the core approach as a task is migrated from a core onto another its dependencies are automatically established. Secondly, agent intelligence is a software abstraction of the sub-task, thereby adding a virtualised layer in the communication stack. This increases the time for communication. The virtual core is also an abstraction of the computing core but is closer to the computing core in the communication stack.

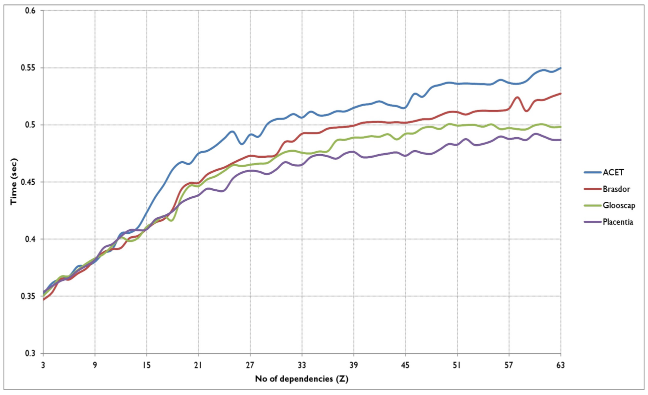
The key observation from the experimental results is that the cost of incorporating intelligence at the task and core levels for automating fault tolerance is in the order of milliseconds. To achieve the full benefits of the approaches, they need to be first incorporated within the task level and then within the core level.

7 Discussion & Conclusions

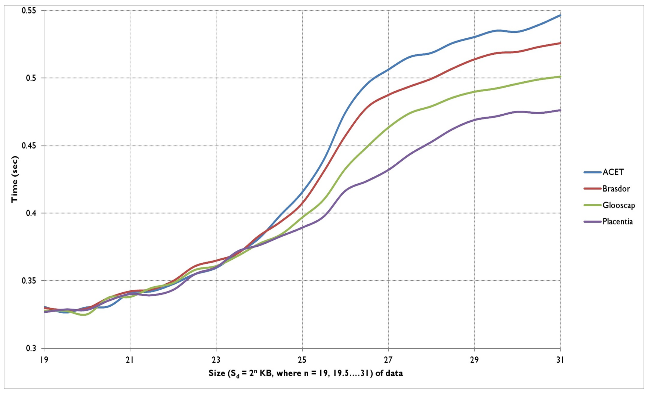
In summary, this paper has presented three approaches towards achieving fault tolerance. In all the approaches, a task to be computed is decomposed into sub-tasks which is then mapped onto the computing cores. The three approaches operate at the middle levels (between the sub-tasks and the computing cores) incorporating agent intelligence. In the first approach, the sub-tasks are mapped onto agents which are



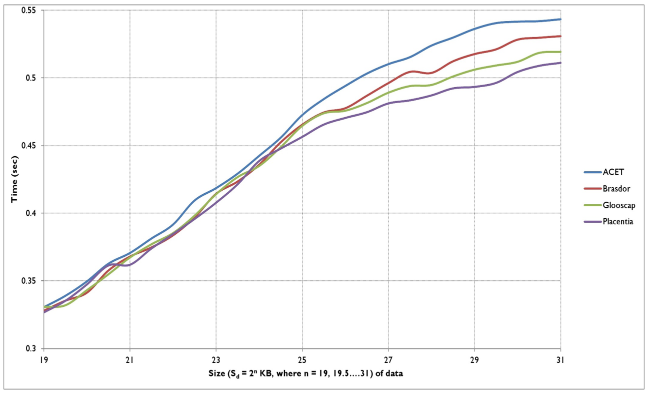
(a) No. of dependencies vs time taken for reinstating execution after failure in the agent intelligent approach



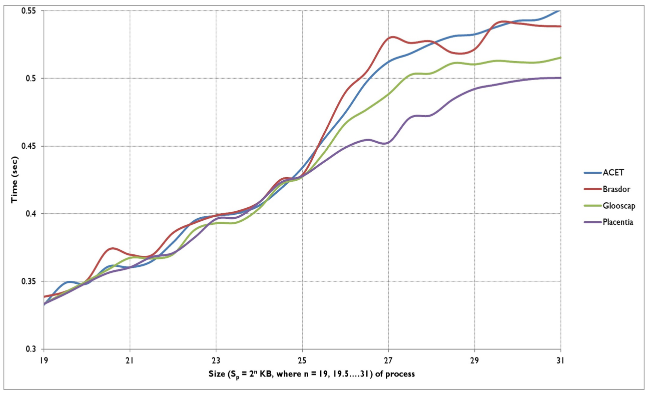
(b) No. of dependencies vs time taken for reinstating execution after failure in the core intelligent approach



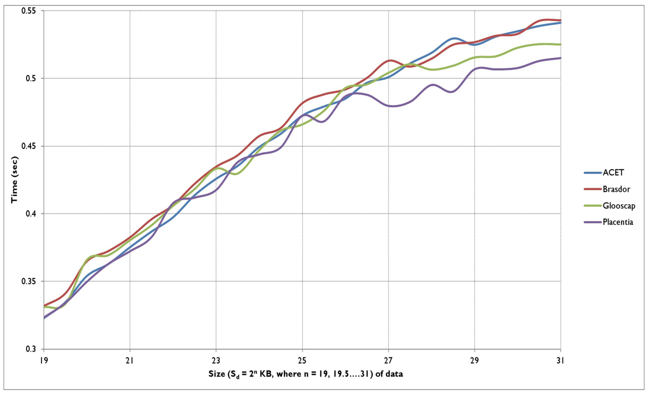
(c) Size of data vs time taken for reinstating execution after failure in the agent intelligent approach



(d) Size of data vs time taken for reinstating execution after failure in the core intelligent approach



(e) Process size vs time taken for reinstating execution after failure in the agent intelligent approach



(f) Process size vs time taken for reinstating execution after failure in the core intelligent approach

Fig. 11: Graphs plotted for the fault tolerant approaches in the second failure scenario

released on the cores. If an agent is notified of a potential core failure during execution of the sub-task mapped onto it, then the agent moves off onto another core thereby automating fault tolerance.

In the second approach the sub-tasks are scheduled on virtual cores, which are an abstraction of the computing cores. If a virtual core anticipates a core failure then it moves off the sub-task on it onto another virtual core, in effect on to another computing core. Both the approaches achieve automation in fault tolerance using intelligence in agents and cores respectively.

However, in the third, a combinative approach, which acts as an umbrella and brings together the concepts of agent and core intelligence. The sub-tasks are mapped onto a set of agents which are released onto virtual cores. When a core failure is anticipated, both the agent and the virtual core are notified. The decision to automate fault tolerance on whether an agent moves off onto another core or whether the virtual core migrates the agent on it onto another core is determined through an arbitration mechanism.

The agent intelligence and core intelligence approaches achieve the same goal of moving away the executing task from a core anticipated to fail but are different in two ways. Firstly, in agent intelligence, the agent is responsible for moving an agent onto another core. In core intelligence, the virtual core is responsible for moving a sub-task onto another core. Secondly, an agent carries information of its dependencies required when it moves off onto another core. The core dependencies do not require to be manually updated.

The three approaches offer minimal human intervening fault tolerant approaches. The foundational concepts of the three approaches are validated on four computer clusters using parallel summation algorithms as a test case. Two failure scenarios are considered in the experimental studies for the first two approaches. The effect of the number of dependencies of a sub-task being executed, the volume of data communicated across cores, the process size and the communication overhead are four factors considered in the experimental studies for determining the performance of the approaches.

Acknowledgments

The authors would like to thank the administrators of the compute resources at the Centre for Advanced Computing and Emerging Technologies (ACET), University of Reading, UK and the Atlantic Computational Excellence Network (ACEnet).

References

[1] F. Cappello, "Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities," International Journal of High Performance Computing Supplications, Vol. 23, Issue 3, pp. 212-226, 2009.

[2] M. R. Varela, K. B. Ferreira and R. Riesen, "Fault-Tolerance for Exascale Systems," Proceedings of the IEEE International Conference on Cluster Computing Workshops and Posters, 2010.

[3] J. Dongarra, P. Beckman et al., "The International Exascale Software Roadmap," International Journal of High Performance Computer Applications, Vol. 25, No. 1, 2011.

[4] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," Journal of Physics: Conference Series, Vol. 78, 2007.

[5] G. Valle, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. Leangsuksun, T. Naughton and S. L. Scott, "A Framework for Proactive Fault Tolerance," in Proceedings of the 3rd IEEE International Conference on Availability, Reliability and Security, pp. 659-664, 2008.

[6] C. Engelmann, G. R. Vallee, T. Naughton and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 252-257, 2009.

[7] Y. Li and Z. Lan, "Current Research and Practice in Proactive Fault Management," International Journal of Computers and Applications, Vol. 29, Issue 4, pp. 408-413, 2007.

[8] S. Chakravorty, C. L. Mendes and L. V. Kale, "Proactive Fault Tolerance in MPI Applications via Task Migration," Proceedings of IEEE International Conference on High Performance Computing, Springer Lecture Notes in Computer Science, Vol. 4297, pp. 485-496, 2006.

[9] J.Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," Proceedings of the 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems, 2007.

[10] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," International Journal of High Performance Computing Applications, Vol. 19 No. 4, pp. 479-493, 2005.

[11] S. Jafar, A. Krings and T. Gautier, "Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing," IEEE Transactions on Dependable and Secure Computing, Vol. 6, No. 1, pp. 32-44, 2009.

[12] T. Saridakis, "Design Patterns for Log-Based Rollback Recovery," Proceedings of the 2nd Nordic Conference on Pattern Languages of Programming, 2003.

[13] E. N. Elnozahy, L. Alvisi, Y. -M. Wang and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message Passing Systems," ACM Computing Surveys, Vol. 34, Issue 3, pp. 375-408, 2002.

[14] D. Sunada, D. Glasco and M. Flynn, "Fault Tolerance: Methods of Rollback Recovery," Stanford University, Technical Report: CSL-TR-97-718, 1997.

[15] A. Litvinova, C. Engelmann and S.L. Scott, "A Proactive Fault Tolerance Framework for High-Performance Computing," Proceedings of the International Conference on Parallel and Distributed Computing and Networks, 2010.

[16] A. Bouteiller, G. Bosilca and J. Dongarra, "Redesigning the Message Logging Model for High Performance," Concurrency and Computation: Practice and Experience, Vol. 22, Issue 16, pp. 2196-2211, 2010.

[17] F. Baude, D. Caromel, C. Delbe and L. Henrio, "A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability," Proceedings of the 11th International Euro-Par Conference on Parallel Processing, pp. 644-653, 2005.

[18] S. Gorender, R. J. de A. Macedo and M. Raynal, "An Adaptive Programming Model for Fault-Tolerant Distributed Computing," IEEE Transactions on Dependable and Secure Computing, Vol. 4, Issue 1, pp. 18-31, 2007.

[19] W. Yurcik and D. Doss, "Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration," IEEE Software, Vol. 18, issue 4, pp. 48-52, 2001.

[20] Z. Lan and Y. Li, "Adaptive Fault Management of Parallel Applications for High-Performance Computing," IEEE Transactions on Computers, Vol. 57, Issue 12, pp. 1647-1660, 2008.

[21] Y. Ren, M. Cukier and W. H. Sanders, "An Adaptive Algorithm for Tolerating Value Faults and Crash Failures," IEEE Transactions on Parallel and Distributed Systems, Vol. 12, Issue, 2, pp. 173-192, 2001.

[22] K. Charoenpornwattana, B. Leangsuksun, A. Tikotekar, G. Vallee and S. Scott, "A Scalable Unified Fault Tolerance for HPC Environments," in Proceedings of the 9th LCI International Conference on High-Performance Clustered Computing, 2008.

[23] Y. Li and Z. Lan, "Exploit Failure Prediction for Adaptive Fault-Tolerance in Cluster Computing," Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid, pp. 531-538, 2006.

[24] A. Steininger, "Dealing With Dormant Faults in an Embedded Fault-Tolerant Computer System," IEEE Transactions on Reliability, Vol. 52, Issue 4, pp. 512-522, 2003.

[25] M. A. Breuer, S. K. Gupta and T. M. Mak, "Defect and Error Tolerance in the Presence of Massive Numbers of Defects," IEEE Design & Test of Computers, Vol. 21, No. 3, pp. 216-227, 2004.

[26] G. K. Saha, "Transient Fault-Tolerance Through Algorithms," IEEE Potential, Vol. 25, No. 5, pp. 25-30, 2006.

[27] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multi-Core Architectures," IEEE Transactions on Dependable and Secure Computing, Vol. 6, No. 2, pp. 135-148, 2009.

[28] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques," IEEE Transactions on Computers, Vol. 52, Issue 9, pp. 1115-1133, 2003.

[29] C. Constantinescu, "Teraflops Supercomputer: Architecture and Validation of the Fault Tolerance Mechanisms," IEEE Transactions on Computers, Vol. 49, Issue: 9, pp. 886-894, 2000.

[30] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. -C. Fabre, J. -C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," IEEE Transactions on Software Engineering, Vol. 16, Issue 2, pp. 166-182, 1990.

[31] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson and U. Gunneflo, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms," IEEE Micro, Vol. 14, Issue 1, pp. 8-23, 1994.

[32] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet and G. Leber, "Integration and Comparison of Three Physical Fault Injection Techniques," Predictably Dependable Computing Systems, Chapter V: Fault Injection, pp. 309-329, 1995.

[33] J. R. Samson, Jr., W. Moreno and F. Falquez, "A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI)," Digest of Papers of the 28th Annual International Symposium on Fault-Tolerant Computing, pp. 162-167, 1998.

[34] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun G. Bosilca, J. P. -Grbovic, J. J. Dongarra, "Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing," International Journal for High Performance Applications and Supercomputing, Vol. 19, No. 4, pp. 465-477, 2005.

[35] C. -H. Yeh, "The Robust Middleware Approach for Transparent and Systematic Fault Tolerance in Parallel and Distributed Systems," Proceedings of the International Conference on Parallel Processing, pp. 61-68, 2003.

[36] J. C. Mourino, M. J. Martin, P. Gonzalez and R. Doallo, "Fault-Tolerant Solutions for a MPI Compute Intensive Application," Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, pp. 246-253, 2007.

[37] X. Yang, Y. Du, P. Wang, H. Fu and J. Jia, "FTPA: Supporting Fault-Tolerant Parallel Computing through Parallel Recomputing," IEEE Transactions on Parallel and Distributed Systems, Vol. 20, No. 10, pp. 1471-1486, 2009.

[38] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-Aware Runtime Strategies for High Performance Computing," IEEE Transactions on Parallel and Distributed Systems , Vol. 20, No. 4, pp. 460-473, 2009.

[39] S. S. Kulkarni and A. Ebnenasir, "Complexity Issues in Automated Synthesis of Failsafe Fault-Tolerance," IEEE Transactions on Dependable and Secure Computing, Vol. 2, No. 3, pp. 201-215, 2005.

[40] A. Ebnenasir, S. S. Kulkarni and A. Arora, "FTSyn: A Framework for Automatic Synthesis of Fault-Tolerance," International Journal on Software Tools for Technology Transfer, Vol. 10, No. 5, pp. 455-471, 2008.

[41] A. Roy-Chowdhury and P. Banerjee, "Algorithm-Based Fault Location and Recovery for Matrix Computations on Multiprocessor Systems," IEEE Transactions on Computers, Vol. 45, No. 11, pp. 1239-1247, 1996.

[42] S. Yajnik and N. K. Jha, "Graceful Degradation in Algorithm-Based Fault Tolerant Multiprocessor Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 2, pp. 137-153 1997.

[43] Z. Chen and J. Dongarra, "Algorithm-Based Fault Tolerance for Fail-Stop Failures," IEEE Transactions on Parallel Distributed Systems, Vol. 19, No. 12, pp. 1628-1641, 2008.

[44] Z. Chen and J. Dongarra, "Algorithm Based Checkpoint Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources," Proceedings of the 20th international conference on Parallel and Distributed Processing, 2006.

[45] L. Yuan, "Generic Fault Tolerant Software Architecture Reasoning and Customization," IEEE Transactions on Reliability, Vol. 55, Issue 3, pp. 421-435, 2006.

[46] Z. Xie, H. Sun and K. Saluja, "A Survey of Software Fault Tolerance Techniques,"

[47] J. M. Smith, "A Survey of Software Fault Tolerance Techniques," Columbia University Computer Science Technical Reports, CUCS-325-88, 1988.

[48] J. C. Ruiz, M. -O. Killijian, J. -C. Fabre and P. Thevenod-Fosse, "Reflective Fault-Tolerant Systems: From Experience to Challenges," IEEE Transactions on Computers, Vol. 52, No. 2, pp. 237-254, 2003.

[49] G. A. Reiss, J. Chang, N. Vachharajani, R. Rangan, D. I. August and S. S. Mukherjee, "Software-Controlled Fault Tolerance," ACM Transactions on Architecture and Code Optimization, Vol. 2, Issue 4, pp. 366-396, 2005.

[50] I. Koren and C. M. Krishna, "Fault Tolerant Systems," Morgan Kaufmann Publisher, 1st Edition, 2007.

[51] A. Aviziensis, "The N-Version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, Vol. 11, Issue 12, pp. 1491-1501, 1985.

[52] D. Avresky, J. Arlat, J. -C. Laprie and Y. Crouzet, "Fault Injection for Formal Testing of Fault Tolerance," IEEE Transactions on Reliability, Vol. 45, Issue 3, pp. 443-455, 1996.

[53] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," IEEE Transactions on Computers, Vol. 44, No. 2, pp. 248-260, 1995.

[54] P. Marinescu and G. Candea, "LFI: A Practical and General Library-Level Fault Injector," Proceedings of the International Conference on Dependable Systems and Networks, Portugal, 2009.

[55] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," IEEE Transactions on Software Engineering, Vol. 24, No. 2, pp. 125-136, 1998.

[56] N. Nakka, A. Agarwal and A. Choudhary, "Predicting Node Failure in High Performance Computing Systems from Failure and Usage Logs," in the Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, pp. 1557-1566, 2011.

[57] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," Proceedings of the 10th international Parallel Processing Symposium, pp. 526-531, 1996.

[58] J. P. Walters and V. Chaudhary, "Replication-Based Fault Tolerance for MPI Applications," IEEE Transactions on Parallel and Distributed Systems, Vol. 20, No. 7, July 2009, pp. 997-1010.

[59] J. P. Walters and V. Chaudhary, "Replication-Based Fault Tolerance for MPI Applications," IEEE Transactions on Parallel and Distributed Systems, Vol. 20, Issue 7, pp. 997-1010, 2009.

[60] A. Moody, G. Bronevetsky, K. Mohor and B. R. de Supinksi, "Design, Modelling, and Evaluation of a Scalable Multi-level Checkpointing System," Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, 2010.

[61] R. Agarwal,. P. Garg and J. Torrellas, "Rebound: Scalable Checkpointing for Coherent Shared Memory," Proceedings of the 38th Annual International Symposium on Computer Architecture, pp. 153-164, 2011.

[62] J. Ho, C. -L. Wang and F. Lau, "Scalable Group-based Checkpoint/Restart for Large-Scale Message-Passing Systems," Proceedings of the 22nd IEEE International Parallel Distributed Processing Symposium, pp. 1-12, 2008.

[63] J. S. Plank, "Diskless Checkpointing," IEEE Transactions on Parallel and Distributed Systems, Vol. 9, Issue 10, pp. 972-986, 1998.

[64] L. A. B. Gomez, N. Maruyama, F. Cappello and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 63-72, 2010.

[65] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello and S. Matsuoka, " Low-Overhead Diskless Checkpoint for Hybrid Computing Systems," Proceedings of the International Conference on High Performance Computing, 2010.

[66] A. Subbiah and D. M. Blough, "Distributed Diagnosis in Dynamic Fault Environments," IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 5, pp. 453-467, 2004.

[67] S. Lee and K. G. Shin, "Optimal and Efficient Probabilistic Distributed Diagnosis Schemes," IEEE Transactions on Computers, Vol. 42, Issue 7, pp. 882-886, 1993.

[68] G. Janakiraman, J. R. Santos and D. Subhraveti, "Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems," Proceedings of the International Conference on Dependable Systems and Networkds, pp. 260-269, 2005.

[69] J. Ansel, K. Arya and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, 2009.

[70] D. Hakkarinen and Z. Chen, "N-Level Diskless Checkpointing," Proceedings of the 11th IEEE International conference on High Performance Computing and Communications, 2009.

[71] P. D. Hough, M. E. Goldsby, and E. J. Walsh, "Algorithm-dependent Fault Tolerance for Distributed Computing," Sandia Report, SAND2000-8219, February 2000.

[72] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. De Turuck, P. Demeester and P. A. Vanrolleghem, "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids," IEEE Transactions on Parallel and Distributed Systems, Vol. 20, Issue 2, February 2009, pp. 180-190.

[73] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun and S. Scott, "A Reliability-Aware Approach for an Optimal Checkpoint/Restart Model in HPC Environments," Proceedings of International Conference on Cluster Computing, pp. 452-457, 2007.

[74] T. Dohi, T. Ozaki and N. Kalo, "Optimal Checkpoint Placement with Equality Constraints," Proceedings of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing, pp. 77-84, 2006.

[75] A. J. Oliner, L. Rudolph and R. K. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," Proceedings of the 20th Annual International Conference on Supercomputing, pp. 14-23, 2006.

[76] A. J. Oliner, L. Rudolph and R. K. Sahoo, "Cooperative Checkpointing Theory," Proceedings of the International Parallel and Distributed Processing Symposium, 2006.

[77] Y. Xiang, Z. Li and H. Chen, "Optimizing Adaptive Checkpointing Schemes for Grid Workflow Systems," Proceedings of the 5th International Conference on Gird and Cooperative Computing Workshops, pp. 181-188, 2006.

[78] S. Chakravorty and L. V. Kale, "A Fault Tolerant Protocol for Massively Parallel Systems," Proceedings of the 18th International Symposium on Parallel and Distributed Processing, 2004.

[79] A. Guermouche, T. Ropars, E. Brunet, M. Snir and F. Cappello, "Uncordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications," Proceedings of the IEEE International Parallel & Distributed Processing Symposium, pp. 989-1000, 2011.

[80] Y. -M. Wang and W. K. Fuchs, "Optimal Message Log Reclamation for Uncoordinated Checkpointing," Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 24-29, 1994.

[81] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez and F. Cappello, "Blocking vs Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI," Proceedings of the ACM/IEEE Conferernce on Supercomputing, Article No. 127, 2006.

[82] Q. Gao, W. Huang, M. J. Koop and D. K. Panda, "Group-Based Coordinated Checkpointing for MPI: A Case Study on InfiniBand," Proceedings of the International Conference on Parallel Processing, 2007.

[83] R. Baldoni, "A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability," Proceedings of the 27th International Symposium on Fault-Tolerant Computing, 1997.

[84] J. Tsai, S. -Y. Kuo and Y. -M. Wang, "Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability," IEEE Transactions on Parallel and Distributed Systems, Vol. 9, Issue 10, pp. 963-971, 1998.

[85] M. Li, D. Goldberg, W. Tao and Y. Tamir, "Fault-Tolerant Cluster Management for Reliable High-Performance Computing," in Proceedings of the 13th International Conference on Parallel and Distributed Computing and Systems, pp. 480-485, 2001.

[86] R. Guerraoui and A. Schiper, "Software-Based Replication for Fault Tolerance," Computer, Vol. 30, Issue 4, pp. 68-74, 1997.

[87] H. Jung, H. Han, H. Y. Yeom and S. Kang, "Athanasia: A User-Transparent and Fault-Tolerant System for Parallel Applications," IEEE Transactions on Parallel and Distributed Systems , Vol. 22, No. 10, pp. 1653-1668, 2011.

[88] J. Cao, Y. Li and M. Guo, "Process Migration for MPI Applications based on Coordinated Checkpoint," Proceedings of the 11th International Conference on Parallel and Distributed Systems, pp. 306-312, 2005.

[89] G. Vallee, C. Morin, J. -Y. Berthou, I. D. Malen and R. Lottiaux, "Process Migration based on Gobelins Distributed Shared Memory," Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002.

[90] T. Boyd, and P. Dasgupta, "Process Migration: A Generalized Approach using a Virtualizing Operating System," Proceedings of the 22nd International Conference on Distributed Computing Systems, pp. 385-392, 2002.

[91] H. Jiang, and V. Chaudhary, "Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems," Proceedings of the 37th Hawaii International Conference on System Sciences, 2004.

[92] T. Maoz, A. Barak and L. Amar, "Combining Virtual Machine Migration with Process Migration for HPC on Multi-Clusters and Grids," Proceedings of the IEEE International Conference on Cluster Computing, pp. 89-98, 2008.

[93] A. D. Blumer, H. Mortveit and C. D. Patterson, "Formal Modelling of Process Migration," Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 104-110, 2007.

[94] R. F. De Mello and L. J. Senger, "A New Migration Model Based on the Evaluation of Processes Load and Lifetime on Heterogeneous Computing Environments," Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, pp. 222-227, 2004.

[95] S. Bertozzi, A. Acquaviva, D. Bertozzi and A. Poggiali, "Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study," Proceedings of Design, Automation and Test in Europe, pp. 15-20, 2006.

[96] M. R. Lyu, X. Chen, and T. Y. Wong, "Design and Evaluation of a Fault-Tolerant Mobile-Agent System," IEEE Intelligent Systems, Vol. 19, Issue 5, pp. 32-38, 2004.

[97] P. Slechta, R. J. Staron, F. P. Matura and K. H. Hall, "Multi-agent Technology for Fault Tolerance and Flexible Control," IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Vol. 36, Issue 5, pp. 700-704, 2006.

[98] S. Pleisch and A. Schiper, "Fault-Tolerant Mobile Agent Execution," IEEE Transactions on Computers, Vol. 52, No. 2, pp. 209-222, 2003.

[99] Ad. L. Almeida, S. Aknine, J. -P. Briot and J. Malenfant, "100," Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, 2006.

[100] M. J. G. C. Mendes, B. M. S. Santos and J. Sa da Costa, "Multi-agent Platform for Fault Tolerant Control Systems," Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, pp. 1321-1326, 2007.

[101] N. Faci, Z. Guessoum and O. Marin, "DimaX: A Fault-Tolerant Multi-Agent Platform," Proceedings of the International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, pp. 13-20, 2006.

[102] The Atlantic Computational Excellence Network (ACEnet) website: http://www.ace-net.ca/

[103] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, 'Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,' Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97-104.

[104] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive MPI," Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958, 2003, pp. 306-322.

[105] L. V. Kale and S. Krishnan, "CHARM++: Parallel Programming with Message-Driven Objects," Parallel Programming using C++ (Eds. G. V. Wilson and P. Lu), MIT Press, 1996, pp. 175-213.

[106] M. L. James, A. A. Shapiro, P. L. Springer and H. P. Zima, "Adaptive Fault Tolerance for Scalable Cluster Computing in Space," International Journal of High Performance Computing Applications, Vol. 23, No. 3, pp. 227-241, 2009.

Blesson Varghese was awarded a PhD in Computer Science from the University of Reading, UK in 2011 on multiple international scholarships. In 2008 he obtained a MSc in Network Centred Computing with distinction again from Reading on the prestigious Felix scholarship. In 2006, he graduated with a first rank from the University of Kerala, India for a four year undergraduate degree in Information Technology. His research interests span across high-performance computing, multi-agent systems, swarm robotic systems and cognitive neuroscience.

Gerard McKee was awarded a BSc in Electronics (Hons) First Class from University of Manchester Institute of Science and Technology (UMIST), UK in 1980. In 1989 he was awarded a PhD also from UMIST. Dr McKee has led the JISC NTI funded project NETROLAB (Networked Robotics Laboratory), and has developed concepts of Visual Acts, Networked Robotics, MARS (modelling and reasoning about modular robotics systems) model, DEIMOS software model implementing MARS model. He has also chaired several international conferences, workshops and symposiums. His research interests include Networked Robotics, Space Robotic Systems, Cooperative Robotics, Artificial Intelligence and Cognitive Intelligence.

Vassil Alexandrov was awarded an MSc in Applied Mathematics from Moscow State University, Russia in 1984 and a PhD in Parallel Computing from Bulgarian Academy of Sciences in 1995. He has held previous positions at the University of Liverpool, UK from 1994-1999 and the University of Reading, UK from 1999-2010. At Reading, he was Professor of Computational Science leading the Computational Science Research Group until September 2010, and the Director of the Centre for Advanced Computing and Emerging Technologies until July 2010. His research interests include Computational Science and High Performance Computing encompassing Parallel, Scalable Algorithms for Advanced Computer Architectures, Monte Carlo methods and algorithms.