International Journal of Computer Theory and Engineering, Vol. 2, No. 5, October, 2010
1793-8201

A Transition from Traditional Checkpointing towards Multi-Agent based Approaches

Gerard McKee, Blesson Varghese and Vassil Alexandrov

***Abstract*—Middleware for parallel computing systems incorporate checkpointing to achieve fault tolerance. Most traditional checkpointing approaches tend to be less dynamic in large scale parallel computing environments. Hence, there arises a need for an adaptive and dynamic approach. The work reported in this paper, proposes a multi-agent based approach for fault tolerance. Five resources namely, the executed problem, parallel computing platform, middleware, hardware abstraction and agents that contribute towards the infrastructure of the proposed approach is considered. The approach is implemented on a computer cluster and experimental results are presented to validate the feasibility of the approach and its contribution towards enhancing fault tolerance.**

***Index Terms*—middleware approach, multi-agent, fault tolerance, parallel computing systems.**

I. Introduction

Research in fault tolerance for parallel computing systems has led to implementations of fault tolerant approaches in three categories, namely the hardware approach, the software approach and the middleware approach [1]. The hardware approach, though expensive to implement, aims to achieve fault free processor networks and is implemented in specific application domains such as space robotic applications or industrial automation machinery.

The software approach, otherwise also referred to as the robust algorithm approach, aims to design algorithms and programs that can run on faulty networks without banking on hardware. The middleware approach on the other hand, aims to add an additional interface or a sandwich layer between hardware and software layers so as to contribute towards fault tolerance. The software and middleware approach are usually adopted in traditional fault tolerance mechanisms.

Middleware approaches are beneficial over other approaches considered above due to three reasons. Firstly, though an additional layer is added to the system topology, fault tolerance is achieved by minimal changes in the hardware and software layers. Secondly, with the existing system topology, middleware layers can draw maximum benefit. Hence, migration to newer platforms will not be frequently required.

One such middleware layer is the Message Passing Interface (MPI), a middleware layer for message-passing designed for massively parallel machines and workstation clusters [2]. MPI implements traditional checkpointing as a strategy for fault tolerance. However, to improve efficiency of checkpointing, variant strategies have been adopted in MPI based research.

Martin & Gonpalves introduces the concept of automatic checkpointing in LAM/MPI middleware [3]. The strategy records the context of an application periodically, identifies failed nodes and restarts MPI processes only on failed nodes, hence allowing continuity of the executing application by taking advantage of the computing done previously.

Yeh proposes DREAM (Dynamic Robust Embedding/Allocation Middleware) based on Robust MPI (R-MPI) as a library component [1]. Chen & Dongarra address challenges in diskless checkpointing by introducing algorithm-based fault tolerance (ABFT) using Fault Tolerant MPI (FT-MPI) [4]. Recovery from failure in the middle of computations is performed by maintaining a checksum relationship.

Walters & Chaudhary address the scalability issue of checkpointing in MPI applications by introducing an asynchronous replication strategy that distributes replication overhead over all participating nodes in the computation [5].

Selvakumar et al tests parallel weather model using fault tolerant MPI comprising a replicated system controller, a node controller and checkpoint server [6]. The fault tolerant version is designed to address single point failures, ensure consistency of checkpoint files and robustness of fault detection hierarchy.

Mourino et al propose two approaches for checkpoint based fault tolerance in computationally intensive applications using MPI [7]. Firstly, segment-level solution, an extension of a checkpoint library for sequential codes. Secondly, variable level solution, a manual solution determined by the programmer that inserts safe points and specifies data to be stored during checkpointing into program code.

Shwe & Aye propose an extension to MPI that consists of two steps to achieve fault tolerance [8]. Firstly, failure diagnosis, a step for detecting the location of a failed component. Secondly, failure recovery, a step towards reassigning tasks of a failed component to fully functional system nodes.

Gerard McKee is Senior Lecturer in Networked Robotics, School of Systems Engineering, University of Reading, Whiteknights Campus, Reading, Berkshire, United Kingdom, RG6 6AY, email: g.t.mckee@reading.ac.uk.

Blesson Varghese is a PhD candidate with the Active Robotics Laboratory, School of Systems Engineering, University of Reading, Whiteknights Campus, Reading, Berkshire, United Kingdom, RG6 6AY, email: b.varghese@student.reading.ac.uk.

Vassil Alexandrov is Professor in Computational Science, School of Systems Engineering, University of Reading, Whiteknights Campus, Reading, Berkshire, United Kingdom, RG6 6AY, email: v.n.alexandrov@reading.ac.uk.

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG

However, checkpointing is challenged by three drawbacks. Firstly, server based checkpointing strategies are subject to single point of failure that tend to be less scalable on complex and heterogeneous environments [5]. Secondly, an attempt to checkpoint a large process involves large overheads and greater time to write the checkpoint to a stable storage system [4][6]. Thirdly, most checkpoint strategies require a cold restart, that is to say, a complete reload of all processes associated with the parallel job [9]. In this case, processors that did not fail might also require a reload of the process executing on it.

Hence, it is apparent that there is a need for an adaptive and dynamic approach for fault tolerance. Multi-agent technology in the context of fault tolerance is favourable for achieving dynamic fault tolerance due to three reasons. Firstly, an agent can be aware of the environment it is situated in. Secondly, an agent can sense hazards that will impair its functioning. Thirdly, an agent can traverse from one location to another if necessary. These three capabilities of agents in a computing environment can be utilized for fault tolerance.

The work reported in this paper aims to incorporate multiagent technology for fault tolerance in parallel computing systems. The agents in the proposed approach demonstrate intelligence within the computing environment. A task to be executed on a computer cluster is decomposed into sub-tasks and mapped onto agents that carry these tasks onto nodes or cores for execution. Resources that come into play in implementing the approach are also considered.

The remainder of the paper is organised as follows. Section 2 considers the five resources that are required as an infrastructure for the approach. Section 3 deals with the implementation details of the proposed approach. Section 4 presents statistic results obtained and calculated from experiments. Section 5 concludes the paper.

II. Resources

To implement the multi-agent fault tolerant approach proposed in this paper five fundamental resources, namely the executed problem, parallel computing platform, middleware, hardware abstraction, intelligent agents that contribute towards the approach need to be considered.

A. Executed Problem

Firstly, the problem to be executed on a parallel computing system needs to be considered. Parallel reduction algorithms, which implement the bottom-up approach of binary trees, are of interest in the context of fault tolerance for multi-agent systems due to two main reasons. Firstly, the computing nodes of a parallel reduction algorithm tend to be critical. The execution of the algorithm stalls or produces an incorrect solution if any node information is lost. Secondly, parallel reduction algorithms are employed in critical applications such as space applications. These applications require fault tolerant distributed systems.

In this paper, parallel summation, an example of parallel reduction is considered. Figure 1 is an illustration of the parallel summation algorithm that will be considered in a later section. The problem of addition is sub-divided between nodes as shown in the diagram, thereby generating sub-problems. These sub-problems are executed on nodes in parallel for a given level, but executed in a sequential manner on nodes between different levels.

B. Parallel Computing Platform

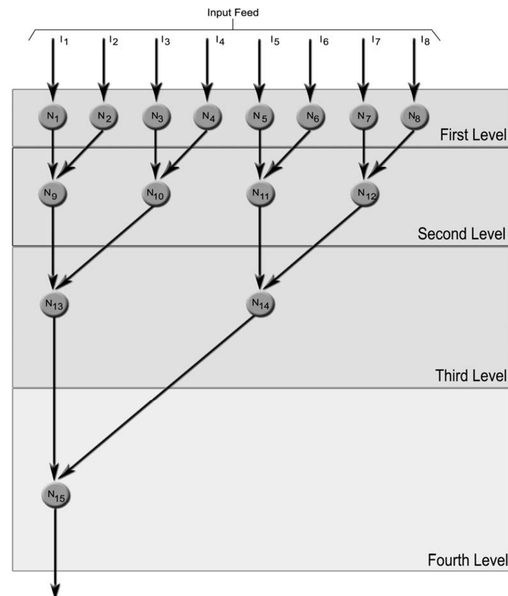

Fig. 1. Illustration of the Parallel Summation Algorithm

Secondly, the computing platform needs to be considered. A computer cluster is chosen as a platform for implementing the multi-agent approach for two reasons. Firstly, a cluster is often characterized by three basic elements, namely a collection of nodes, a network connecting these nodes and a facility to access and share information between the nodes [10], which are comparatively simpler elements to handle when compared to other parallel computing infrastructures. Secondly, existing middleware for clusters, namely Message Passing Interface (MPI) [2] provide standard and portable programming interfaces.

The cluster used for the research reported in this paper is one among the high performance computing resources available at the Centre for Advanced Computing and Emerging Technologies (ACET), University of Reading, United Kingdom [11][12]. The cluster is primarily used for the purpose of teaching and performing multi-disciplinary research. The cluster consists of a head node and 33 compute nodes. All nodes in the cluster are Intel Pentium 4 CPUs connected via a Gigabit ethernet switch and communicate via the standard TCP protocol.

C. Middleware

When you submit your final version, after your paper has been accepted, prepare it in two-column format, including figures and tables. Thirdly, the middleware appropriate for the approach needs to be considered. The cluster-based implementations reported in this paper are based on MPI, a standard application programming interface used for parallel and/or distributed computing. Open MPI [13][14] version 1.3.3, an open source implementation of MPI 2.0 is employed on the cluster. An important feature of MPI 2.0, dynamic process creation and management, is of potential for exploration in the context of swarm-array computing.

The MPI dynamic process model permits the creation and management of a set of processes both when an MPI application begins and after the application has started. The management of newly created processes includes cooperative termination of a process, communication between newly created processes and existing MPI application, and establishing communication between two independent processes. MPI_COMM_SPAWN is used to create a new MPI process and establish communication from an existing MPI application. On the other hand, MPI_COMM_ACCEPT and MPI_COMM_CONNECT can be used to establish communication between two independent processes. More information on the dynamic process model of MPI can be obtained from [2][15].

Since MPI gives control over the process being executed rather than the processor on which a process is being executed, it is appropriate to implement the multi-agent based approach using MPI.

D. Hardware Abstraction

When you submit your final version, after your paper has been accepted, prepare it in two-column format, including figures and tables.

Fourthly, an abstraction of the hardware resource needs to be considered. The hardware resource layer comprises physical nodes of the cluster and is connected via a switch, thereby forming a fully connected mesh topology. However, the abstracted layer is obtained when the physical nodes are abstracted as logical nodes. This is possible by implementing rules/policies. The policies are such that a process executing a sub-task of a problem can only communicate with a vertically, horizontally or diagonally adjacent process, effectively leading to a grid topology on the abstracted layer. For example, nine nodes of a computer cluster forming a fully connected mesh topology in figure 2 is abstracted to a grid topology in the abstraction layer.

E. Agents

Fifthly, agents that contribute towards fault tolerance need to be considered. The parallel summation task to be executed on the cluster is decomposed into sub-tasks and mapped onto agents that carry these tasks onto nodes or cores for execution. The agent and the sub-problem are independent of each other; in other words, the agents only carry the sub-tasks or act as a wrapper around the sub-task independent of the operations performed by the task.

The agents in the proposed approach demonstrate intelligence within the computing environment in four different ways. Firstly, an agent is aware of its environment comprising nodes or cores on which it can carry a task onto, other agents in its vicinity and agents with which it interacts or shares information. Secondly, an agent can situate itself on a node or core that may not fail soon and can provide necessary and sufficient consistency in executing the task. Thirdly, an agent can predict core failures by consistent monitoring (for example, power consumption and heat dissipation of the cores can be used to predict failures). Fourthly, an agent is capable of gracefully shifting from one core to another, without causing interruption to the state of execution, and notifying other interacting agents in the system when a core on which a subtask being executed is predicted to fail.

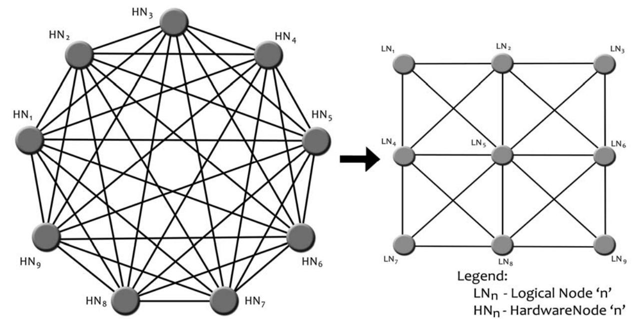

Fig. 2. Mapping hardware nodes to logical nodes

III. Implementation

The parallel summation algorithm as shown in figure 1 works in four sequential levels. The first level comprising nodes $N_1 - N_8$ receives a live input feed of data. The second level comprising nodes $N_9 - N_{12}$ receives data from the first level, adds the data received and yields the result to the third level nodes N_{13} and N_{14}. The fourth level, adds data received from the third level nodes and produces the final result.

For a given time step, every node in a level operates in parallel. Each node is characterized by input dependencies (process or processor a node is dependent on for receiving an input), output dependencies (process or processor a node yields data to as output) and data contained in the node. The first level nodes have one input dependency and one output dependency. For instance, node N_1 has one input dependency I_1 and node N_9 as its output dependency. However, the second, third and fourth levels have two input dependencies and one output dependency. For instance, node N_{13} of the third level has nodes N_9 and N_{10} as input dependencies and node N_{15} as output dependency. The data contained in a node is either the input data for the first level nodes or a calculated value (sum of two values in the case of a parallel summation algorithm) stored within a node.

The agents on the abstracted layer are created such that they carry input and output dependencies and data. Since, parallel summation is relatively less complex when compared to other computational algorithms; the agents carry little information and have only few dependencies.

Each process executing on a node also gathers some sensory information to predict whether a node is likely to fail, on similar lines to proactive fault tolerance. In the implementation presented in this paper node temperatures are simulated. When the temperature of a node rises beyond a threshold, the process executing on that node predicts a failure and hence spawns a process on an adjacent core in the abstracted layer. The agent on the abstracted core expected to fail shifts to the adjacent core on which the new process was spawned. The dependency information carried by the agent that was shifted to the new core is employed to reinstate the state of execution of the algorithm. The data for summation contained in the agent, either obtained from a previous level or a calculated value to be yielded to the next level, ensures that information is not lost and does not affect the final solution in critical applications.

Though a preliminary implementation model was achieved,

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG

TABLE I OBTAINED & CALCULATED VALUES OF TIME FOR THE MULTI-AGENT APPROACH

N_n / TR	TN_n (sec)						MTN_n (sec)	MTLp (sec)	
	1	2	3	4	5	6			
n=9	0.346	0.325	0.378	0.314	0.328	0.346	0.340	p=2	0.345
n=10	0.365	0.369	0.346	0.325	0.369	0.328	0.350		
n=11	0.368	0.334	0.368	0.345	0.355	0.335	0.351		
n=12	0.315	0.368	0.339	0.324	0.368	0.324	0.340		
n=13	0.365	0.346	0.328	0.365	0.336	0.353	0.349	p=3	0.343
n=14	0.342	0.346	0.314	0.346	0.342	0.328	0.336		
n=15	0.336	0.365	0.346	0.387	0.365	0.346	0.358	p=4	0.358

it was observed that MPI was not the most appropriate middleware for implementing the multi-agent approach. When an agent predicted a node failure, a new process had to be dynamically created on an adjacent node that was not predicted to fail, hence allowing the agent on the node predicted to fail to transfer control onto the agent on the newly created process. For this, MPI_Comm_spawn, MPI_Comm_connect and MPI_Comm_accept were required. Since some of these functionalities provided unstable results on the cluster used for implementation, a workaround had to be sought. Hence the process on the new node onto which the agent transferred was created during the initialization of the program and ran on the cluster as a dummy process until it came to play.

IV. RESULTS

A few tests were carried out to note TN_n, the time taken by an agent to transfer from a node N_n predicted to fail onto an adjacent node in the abstracted layer and re-establish all process dependencies for seamless execution. Nodes N_9 – N_{15} as shown in figure 1 are the computational nodes of the parallel summation algorithm, and hence only considered for the calculation of TN_n. Six different trial runs were performed to gather the statistic.

Further, MTN_n, the mean time of TN_n for a particular node was calculated. This metric yields information on the mean time taken by an agent to transfer from a node N_n predicted to fail onto an adjacent node in the abstracted layer and re-establish all process dependencies for seamless execution. MTN_n is calculated as follows:

$$MTN_n = \left(\sum_{TR=1}^{6} T_n \right) \Big/ 6, n = 9...15 \quad (1)$$

MTL_p, the mean time of an agent transfer for all nodes predicted to fail in a level of the parallel summation algorithm onto an adjacent node in the abstracted layer was calculated. Nodes N_9 and N_{12} are used in level 2, while N_{13} and N_{14} in level 3 and N_{15} in level 4. MTL_p is calculated as follows:

$$MTL_2 = \left(\sum_{n=9}^{12} MT_{Nn} \right) \Big/ 4 \quad (2)$$

$$MTL_3 = \left(\sum_{n=13}^{14} MT_{Nn} \right) \Big/ 2 \quad (3)$$

$$MTL_4 = MTN_{15} \quad (4)$$

MTN_N, the mean time of agent transfer for all computational nodes in the parallel summation algorithm onto an adjacent node in the abstracted layer was calculated. This value can be calculated as the mean time of all MTN_n of the computational nodes or the mean time of all MTL_p of the computational levels. MTN_N is calculated as follows:

$$MTN_N = \left(\sum_{n=9}^{15} MT_{Nn} \right) \Big/ 7 \quad (5)$$

or

$$MTN_N = \left(\sum_{p=2}^{4} MT_{Lp} \right) \Big/ 3 \quad (6)$$

Table 1 shows the statistics of the metrics considered above for each computational node (N_9 – N_{15}). Each row in the table is for a specific computational node, Nn, n = 9…15. The time TNn obtained from the experiments and MTL_p are calculated and shown in the table.

The mean time of all the computational nodes MTN_N is calculated as 0.349 sec. This statistic reveals that the time taken for reinstating execution is negligibly small. However, if traditional methods for fault tolerance were used it is more likely that the time taken for reinstating execution would have been greater when compared to the values obtained in the multi-agent approach.

Hence, the preliminary results obtained through simple experiments are promising that the multi-agent approach proposed in this paper can be effectively implemented in parallel computing systems for fault tolerance.

V. CONCLUSIONS

In this paper, a multi-agent fault tolerance approach for parallel computing systems has been proposed. This approach opens new avenues of research and aims to address

challenges posed by traditional checkpointing strategies. Five fundamental resources, namely the executed problem, parallel computing platform, middleware, hardware abstraction and agents that contribute towards realizing the approach are considered. The method aims to map subtasks decomposed from a task to be executed onto agents that carry these subtasks onto nodes or cores for execution. The agents in the proposed approach demonstrate intelligence within the computing environment. Preliminary results indicate that the approach is prospective for achieving fault tolerance.

Future work will aim to analyse the proposed approach using other metrics. The approach will also be implemented to extend it for multi-node failure as against single node failure implemented in this paper. Efforts will also be made to explore other middleware libraries that will prove useful for implementing the multi-agent approach.

References

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

[2] W. Gropp, E. Lusk and A. Skjullum, "Using MPI-2: Advanced Features of the Message Passing Interface" MIT Press, 1999.

[3] A. S. Martins and R. A. L Gonpalves, "Implementing and Evaluating Automatic Checkpointing" in the Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2007, pp. 1-8.

[4] Z. Chen and J. Dongarra, "Algorithm-based Fault Tolerance for Fail-Stop Failures" in the IEEE Transactions on Parallel and Distributed Systems, Vol. 19, Issue 12, December 2008, pp. 1628-1641.

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

[6] A. D. Selvakumar, P. M. Sobha, G. C. Ravindra and R. Pitchiah, "Design, implementation and Performance of Fault-Tolerant Message Passing Interface (MPI)" in the Proceedings of the 7th International Conference on High Performance Computing and Grid in Asia Pacific Region, 2004, pp. 120-129.

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

[8] T. Shwe and W. Aye, "A Fault Tolerant Approach in Cluster Computing System" in the Proceedings of the 5th international Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008, pp. 149-152.

[9] X. Yang, Y. Du, P. Wang, H. Fu and J. Jia, "FTPA: Supporting Fault-Tolerant Parallel Computing through Parallel Recomputing" in the IEEE Transactions on Parallel and Distributed Systems, Vol. 20, Issue 10, October 2009, pp. 1471-1486.

[10] J. D. Sloan, "High Performance Linux Cluster with OSCAR, Rocks, openMosix & MPI" O'Reilly, 2005.

[11] Center for Advanced Computing and Emerging Technologies (ACET) website: www.acet.reading.ac.uk

[12] High Performance Computing at ACET website: http://hpc.acet.rdg.ac.uk/

[13] OpenMPI website: http://www.open-mpi.org/

[14] 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" in the Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97-104.

[15] MPI Tutorial: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

Dr. Gerard McKee is Senior Lecturer in Networked Robotics in the School of Systems Engineering, a Chartered IT Professional (CITP) and Fellow of the British Computer Society (FBCS). He leads the Active Robotics Laboratory which conducts research in the areas of space and networked robotics. His primary research interest is in the area of modular distributed robot architectures and has included the development of networked, space and online robot systems incorporating modular distributed components networked together via a query-based localisation model to create higher-order functional architectures. His primary teaching is in the area of robotics and artificial intelligence.

Dr McKee is an international leader in Networked Robotics, having conducted research and published papers in the area since the early 1990s. He has been a Visiting Research Scholar with the Man-Machine Systems Group at the Jet Propulsion Laboratory (JPL), Pasadena, USA, and Visiting Researcher at the MIT Field Robotics Laboratory, Boston, USA. He has co-edited a special issue of the Journal of Autonomous Robots (November 2003) on the topic of Internet and Online Robots, and has recently given key-note presentations on networked robotics at the International Conference on Informatics in Control, Automation and Robotics (ICINCO-06), the 3rd Latin American Association Symposium (LARS 2006), and the First International Conference on Robotics Communication and Coordination (RoboComm 2007). He has also contributed to workshop, including the IEEE R&A Conference workshop on Omniscient Space: Robot Control Architectures Geared toward Adapting to Dynamic Environments (Roma, Italy, April 2007).

Mr. Blesson Varghese is currently a PhD candidate at the University of Reading, UK. He was the recipient of the the Felix Scholarship 2007 to pursue postgraduation studies and received his MSc in Network Centered Computing from the University of Reading in 2008. He graduated as the gold medalist in B.Tech Information Technology from Kerala University, India in 2006. He has over 30 publications in journals and conference proceedings. He has won several awards for his papers and posters at conferences.

His current research spans across disciplines such as swarm robotics, autonomic computing and Parallel computing that has resulted in a novel concept for fault tolerance in high performance computing systems referred to as 'Swarm-Array computing'. Other areas of his research include multi-agent systems, space robotic applications, distributed algorithms and distributed computing.

Prof. Vassil Alexandrov is a Professor in Computational Sciences at the School of Systems Engineering, Director of the Centre for Advanced Computing and Emerging Technologies and Head of Research of PEDAL Laboratory at the University of Reading, UK. He has obtained his MSc in Applied Mathematics from Moscow State University in 1984 and his PhD in parallel computing from the Institute for Parallel Processing at the Bulgarian Academy of Sciences in 1995.

His main interests are in the area of simulation and modelling of complex systems, parallel scalable algorithms, Collaborative, Cluster and Grid computing and using the advances in the above mentioned areas for efficiently solving large scale scientific and industrial problems. He participates in national and international projects in the area of Collaborative and Grid Computing and e-learning. He also collaborates with Intel and IBM in the areas of Collaborative and High End Computing and with Oak Ridge National Laboratory, Supercomputing Centre in Barcelona, Daresbury Laboratory, IPP - Sofia, SZTAKI - Budapest, Emory University and University of Tennessee in the area of Collaborative, Cluster and Grid computing.

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG