FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems

Rajagopal Subramaniyan, Vikas Aggarwal, Adam Jacobs, and Alan D. George
{subraman, aggarwal, jacobs, george}@hcs.ufl.edu
High-performance Computing and Simulation (HCS) Research Laboratory
Department of Electrical and Computer Engineering, University of Florida
Gainesville, Florida, 32611-6200
Phone: 352.392.9046   Fax: 352.392.8671

Abstract - Ever-increasing demands of space missions for data returns from their limited processing and communications resources have made the traditional approach of data gathering, data compression, and data transmission no longer viable. Increasing on-board processing power by providing high-performance computing (HPC) capabilities using commercial-off-the-shelf (COTS) components is a promising approach that significantly increases performance while reducing cost. However, the susceptibility of COTS components to single-events upset (SEU) is a concern demanding fault-tolerant system infrastructure. Among the components of this infrastructure, message-passing middleware based upon the Message Passing Interface (MPI) standard is essential, so as to support and provide a nearly effortless transition for earth and space science applications in MPI from ground-based computational clusters to HPC systems in space. In this paper, we present the design of a fault-tolerant MPI-compatible middleware for embedded cluster computing known as FEMPI (Fault-tolerant Embedded MPI). We also present preliminary performance results with FEMPI on a COTS-based, embedded cluster system prototype.

Keywords: Message-passing middleware, Fault tolerance, Embedded clusters, Parallel computing, Space systems

1. Introduction

Space missions involving science and defense ventures have ever-increasing demands for data returns from their resources. The traditional approach of data gathering, data compression and data transmission is no longer viable due to vast amounts of data involved. Over the past few decades, there have been several research efforts to make HPC systems available in space. The idea has been to provide sufficient on-board processing power to support a plethora of space and earth exploration and experimentation missions. Such efforts have led to increasingly powerful computers embedded in the spacecraft and, more recently, to the idea of using COTS components to provide HPC in space. NASA's New Millennium Program (NMP) office has commissioned the development of a COTS embedded cluster for space missions [1]. The new Dependable Multiprocessor technology will provide variety of features including: the ability to rapidly infuse future commercial technology into standard COTS payload; a standard development/runtime environment familiar to scientific application developers; and a robust management service to mitigate the effects of space radiation hazards on COTS components.

Fault-tolerant system functions are required to manage the resources and improve availability of the system in space. However, the resources available in a space platform are much less powerful than traditional HPC systems on earth. In order to enable applications to meet high-availability and high-reliability requirements, several approaches can be followed including incorporating fault-tolerant features directly into the applications, developing specialized fault-tolerant hardware, making use of and enhancing the fault-tolerant features of operating systems, and developing application-independent middleware that would provide fault-tolerant capabilities. Among these options, developing application-independent middleware has the minimal intrusion in the system and can support any general application including legacy applications that fall into the umbrella of the corresponding middleware model.

In the past decade, MPI has emerged as the de-facto standard for development and execution of high-performance parallel applications. By its nature as a communication library facilitating user-level communication and synchronization amongst a group of processes, the MPI library needs to maintain global awareness of the processes that collectively constitute a parallel application. An MPI library consequently emerges as a logical and suitable place to incorporate selected fault-tolerant features. However, fault-tolerance is absent in both the MPI-1 and MPI-2 standards [2]. To the best of our knowledge, no satisfactory products or research results offer an effective path to providing scalable computing applications for embedded cluster systems with effective fault-tolerance. In this paper, we present the design and analyze the characteristics of FEMPI, a new lightweight, fault-tolerant message passing middleware for clusters of embedded systems. The scope of this paper is focused upon a presentation of the design of FEMPI and preliminary performance results with it on a COTS-based, embedded cluster system prototype for space. To highlight the compatibility of FEMPI across platforms and permit performance comparisons with conventional MPI middleware, we also provide results from experiments on an Intel Xeon cluster.

The remainder of this paper is organized as follows. Section 2 provides background on several existing fault-tolerant MPI implementations for traditional general-purpose HPC systems. Section 3 discusses the architecture and design of FEMPI, while performance results are presented in Section 4. Section 5 concludes the paper and summarizes insight and directions for future research.

2. Background and Related Research

In this section, we provide an overview of MPI and its inherent limits relative to fault-tolerance. Also included is a brief summary of existing tools that provide fault-tolerance to MPI, albeit primarily targeting conventional, resource-plentiful HPC systems instead of embedded, mission-critical systems that are the emphasis of our work.

2.1 Limitations of MPI Standard

The MPI forum released the first MPI standard, MPI-1, in 1995, with drafts provided over the previous 18 months. The primary goals of the standard in this release were high performance and portability. Achieving reliability typically includes utilization of additional resources and methodologies. This additional utilization conflicts with the goal of high performance and supports MPI Forum's decision for limited reliability measures. Emphasis on high performance thus led to a static process model with limited error handling. The success of an MPI application is guaranteed only when all constituent processes finish successfully. "Failure" or "crash" of one or more processes leads to a default application termination procedure which is in fact required standard behavior. Subsequently, current designs and implementations of MPI suffer from inadequacies in various aspects to providing reliability.

Fault Model: MPI assumes a reliable communication layer. The standard does not provide methods to deal with node failures, process failures, and lost messages. MPI also limits faults in the system to incorrect parameters in function calls and resource errors. This coverage of faults is incomplete and insufficient for high-scale parallel systems and mission-critical systems that are affected by transient faults such as SEUs.

Fault Detection: Fault detection is not defined by MPI. The default mode of operation of the MPI treats all errors as fatal and terminates the entire parallel job. MPI provides for limited fault notification in the form of return codes from the MPI functions. However, critical faults, such as process crashes, may preempt functions from returning these return codes to the caller.

Fault Recovery: MPI provides users with functions to register error-handling callback functions. These callback functions are invoked by the MPI implementation in the event of an error in MPI functions. Callback functions are registered on a per-communicator (communication context defined in MPI for groups of processes) basis and do not allow per function-based error handlers. Callback functions provide limited capability and flexibility.

The MPI Forum released the MPI-2 standard in 1998, after a multi-year standards process. A significant contribution of MPI-2 is Dynamic Process Management (DPM), which allows user programs to create and terminate additional groups of processes on demand. DPM may be used to compensate for the loss of a process while MPI I/O can be used for checkpointing the state of the applications. However, the lack of failure detection precludes the potential for such added reliability.

2.2 Fault-tolerant MPI Implementations for Traditional Clusters

Several research efforts have been undertaken to make MPI more reliable for traditional HPC systems. This section briefly introduces some of these efforts and analyzes their approaches in providing a reliable MPI middleware.

MPICH-V [3] from University of South Paris, France, is an MPI environment that is based upon uncoordinated checkpointing/rollback and distributed message logging. MPICH-V suffers from single points of failure and is only suitable for Master/Worker types of MPI applications. Starfish [4] from the Technion University, Israel, is an environment for executing dynamic MPI-2 programs. Every node executes a starfish daemon and many such daemons form a process group using the Ensemble group communication toolkit [5]. The daemons are responsible for interacting with clients, spawning MPI programs and tracking and recovering from failures.

CoCheck [6] from the University of Germany, Munich, is a checkpointing environment primarily targeted for process migration, load balancing, and stalling long-running applications for later resumption. CoCheck extends the single process checkpoint mechanisms to a distributed message-passing application. Egida [7] from the University of Texas, Austin, is an extensible toolkit to support transparent rollback recovery. Egida is built around a library of objects that implement a set of functionalities that are the core of all log-based rollback recovery. Egida has been ported to MPICH [8], a widely used and free implementation of MPI. Implicit FT-MPI [9] from the University of Cyprus takes an approach similar to CoCheck but is a more stripped-down version. Implicit FT-MPI targets only the master/slave model. Implicit FT-MPI is very simple in terms of features available and suffers from single points of failure. LAM/MPI [10], an implementation of MPI from Indiana University and Ohio Supercomputing Center, has some fault-tolerance features built into it. Special functions such as *lamgrow* and *lamshrink* are used for dynamic addition and deletion of hosts. A fail-stop model is assumed and, whenever a node fails, it is detected as dead and the resource manager removes the node from the host lists.

FT-MPI [11] from University of Tennessee, Knoxville, attempts to provide fault-tolerance in MPI by extending the MPI process states and communicator states from the simple {valid, invalid} as specified by the standard to a range of states. A communicator is an important scoping and addressing data structure defined in the standard that defines a communication context, and a set of processes in the context. The range of communicator states specified by FT-MPI helps the application with the ability to decide how to alter the communicator, its state and the behavior of the communication between intermediate states on occurrence of a failure. FT-MPI provides for graceful degradation of applications but has no support for transparent recovery from faults. The design concept of FEMPI is also largely based upon FT-MPI, although there are significant differences as FEMPI is designed to target embedded, resource-limited, and mission-critical systems where faults are more commonplace such as payload processing in space.

3. Design Overview of FEMPI

As described in the previous section, several efforts have been undertaken to develop a fault-tolerant MPI. However, all the designs described previously target conventional large-scale HPC systems with sufficient system resources to cover the additional overhead for fault tolerance. More importantly, very few among those designs have been successfully implemented and are mature enough for practical use. Also, the designs are quite heavyweight primarily because fault tolerance is based on extensive message logging and checkpointing. Many of the designs are also based on centralized coordinators that can incur severe overhead and become a bottleneck in the system. An HPC system in space requires a reliable and lightweight design of fault-tolerant MPI. Among those that exist, FT-MPI from University of Tennessee [11] was viewed to be the one with the least overhead and is the most mature design in terms of realization. However, FT-MPI is built atop a metacomputing system called Harness that can be too heavy for embedded systems to handle.

For the design of FEMPI, we try to avoid design and performance pitfalls of existing HPC tools for MPI but leverage useful ideas from these tools. FEMPI is a lightweight design in that it does not depend upon any message logging, specialized checkpointing, centralized coordination or other large middleware systems. Our design of FEMPI closely resembles FT-MPI. However, the recovery mechanism in FEMPI is different in that it is completely distributed and does not require any reconstruction of communicators. FT-MPI requires the reconstruction of communicators on a failure for which the system enters an election state and a voting-based leader selection is performed. The leader is responsible for distributing the new communicator to all the nodes in the system.

3.1 FEMPI Architecture

Performance and fault-tolerance are in general competing and often conflicting goals in HPC. Providing fault-tolerant services in software inevitably adds extra processing overhead and increases system resource usage. With FEMPI, we try to address both fault tolerance and performance and where conflicting then fault tolerance is given priority. We address fault-tolerance issues in both the MPI-1 and MPI-2 standards, with attention to key application classes, fault-free overhead, and recovery strategies.

Fault tolerance is provided through three stages including detection of a fault, notification of the fault, followed by recovery from the fault. In order to reduce development time and to ensure reliability, FEMPI is built on top of a commercial high-availability (HA) middleware called Self-Reliant (SR) from GoAhead Inc. whose services are used to provide detection and notification capabilities. SR allows processes to heartbeat through certain fault handlers and hence has the potential to detect the failure of processes and nodes (enabled by the Availability and Cluster Management Services in Figure 1). CMS manages the physical nodes or instances of HA Middleware, while AMS manages the logical representation of these and other resources in the availability system model. The fault notification service for application processes is developed as an extension to SR. SR also guarantees reliable communication via network interface failover capabilities and in-order delivery of messages between the nodes in the system through Distributed Messaging Service (DMS), a part of SR.

Figure 1 shows the architecture of FEMPI. The application is required to register with the Control Agent in each node, which in turn is responsible for updating the health status of the application in that node to a central Control Process. The Control Process is similar to a system manager, scheduling applications to various data processing nodes. In the actual mission, the Control Process will execute on a radiation-hardened, single-board computer that is also responsible for interacting with the main controller for the entire spacecraft. Although such system controllers are highly reliable components, they can be deployed in a redundant fashion for highly critical or long-term missions with cold or hot sparing. The details of the Control Process are beyond the scope of this paper.

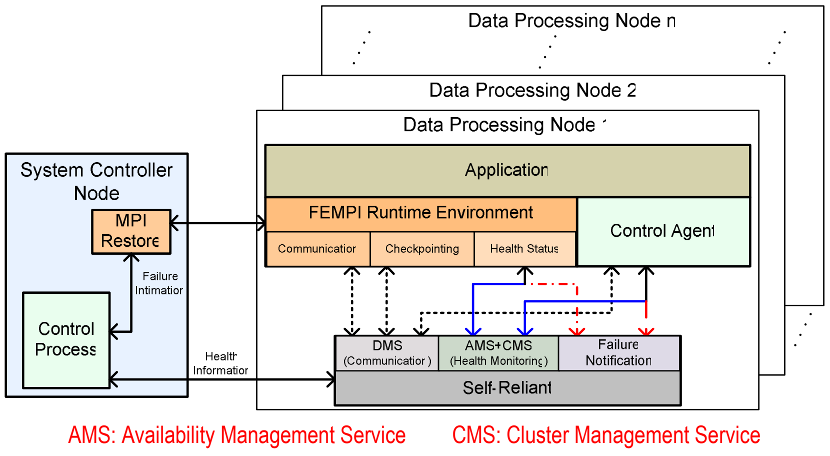


Figure 1. Architecture of FEMPI

With MPI applications, failures can be broadly classified as process failures and network failures. FEMPI ensures reliable communication (reducing the chances of network failures) with all low-level communication through DMS. A process failure in conventional MPI implementations causes the entire application to crash, whereas FEMPI avoids application-wide crashes when individual processes fail. MPI Restore, a component of FEMPI, resides on the System Controller and communicates with the Control Process to update the status of nodes. On a failure, MPI Restore notifies all other MPI processes regarding the failure via DMS. The status of senders and receivers (of messages) are checked in FEMPI before communication to avoid trying to establish communication with failed processes. If the communication partner (sender or receiver) fails after the status check and before communication, then a timeout-based recovery is used to recover out of the MPI function call.

With our initial design of FEMPI as described in this paper, we focus only on selected baseline functions of the MPI standard. The first version of FEMPI includes 19 baseline functions shown in Table 1, of which four are setup, three are point-to-point messaging, five are collective communication, and seven are custom data-definition calls. The function calls were selected based upon profiling of popular and commonly used space science kernels such as Fast Fourier Transform, LU Decomposition, etc. In our initial version of FEMPI, we also focus only on blocking and synchronous communications, which is the dominant mode in many MPI applications. Blocking and synchronous communications require the corresponding calls (e.g. send and receive) to be posted at both the sender and receiver processes to either process to continue further execution.

Table 1. Baseline MPI functions for initial version of FEMPI

Function	MPI Call	Type	Purpose
Initialization	*MPI_Init*	Setup	Prepares system for message-passing functions.
Communication Rank	*MPI_Comm_rank*	Setup	Provides a unique node number for each node.
Communication Size	*MPI_Comm_size*	Setup	Provides the number of nodes in the system.
Finalize	*MPI_Finalize*	Setup	Disable communication services.
Send	*MPI_Send*	P2P	Send data to a matching receive.
Receive	*MPI_Recv*	P2P	Receive data from a matching send.
Send-Receive	*MPI_Sendrecv*	P2P	Simultaneous send and receive between two nodes i.e. send, receive using the same buffer.
Barrier Synchronization	*MPI_Barrier*	Collective	Synchronize all the nodes together.
Broadcast	*MPI_Bcast*	Collective	"Root" node sends same data to all other nodes.
Gather	*MPI_Gather*	Collective	Each node sends a separate block of data to "root " node to provide an all-to-one scheme
Scatter	*MPI_Scatter*	Collective	"Root" node sends a different set of data to each node providing a one-to-allscheme
All-to-All	*MPI_Allgather*	Collective	All nodes share data with all other nodes in the system.
Datatype	*MPI_Type_**	Custom	Seven functions to define custom data types useful for complicated calculations

3.2 Recovery Modes

Presently, two different recovery modes have been developed in FEMPI, namely IGNORE and RECOVER. In the IGNORE mode of recovery, when a FEMPI function is called to communicate with a failed process, the communication is not performed and MPI_PROC_NULL (meaning the process location is empty) is returned to the application. Basically, the failed process is ignored and computation proceeds without any effect of the failure. The application can either re-spawn the process through the control process or proceed with one less process. The MPI communicators are left unchanged. The IGNORE mode is useful for applications that can execute with reduced number of processes while the failed process is being recovered. With RECOVER mode of recovery, the failed process has to be re-spawned back to its healthy state either on the same or a different processing node. When a FEMPI function is called to communicate with a failed process, the function call is halted until the process is successfully re-spawned. When the process is restored, the communication call is completed and the control is returned back to the application. Here again, the MPI communicators are left changed. The RECOVER mode is useful for applications that cannot execute with any less number of nodes than when they started execution. In future, we plan to develop a third mode of recovery namely REMOVE. With the REMOVE mode of recovery, when a process fails, it is removed from the process list. REMOVE would be useful for cases when processes have irrecoverably failed while running applications that cannot proceed with any failed process in the system.

4. Experimental Results

For the current research phase of the NASA's NMP project, a prototype system designed to emulate the features of a typical satellite system has been developed at Honeywell Inc. and the University of Florida. The prototype hardware consists of a collection of single-board computers, some augmented with FPGA coprocessors, redundant Ethernet switches, and a development workstation acting as satellite controller. Six Orion Technologies COTS Single-Board Computers (SBCs) are used to mirror the data processor boards to be featured in the flight experiment and also to emulate the functionality of the radiation-hardened components under development. Each board includes a 650MHz IBM 750fx PowerPC, 128MB of high-speed DDR SDRAM, dual Fast Ethernet interfaces, dual PCI mezzanine card slots, and 256MB of flash memory and runs MontaVista Linux. Of the six boards, two are used as the control nodes (primary and backup), one is used as central data store, and three as data processing nodes that we will exercise to show the performance of FEMPI. In this paper, we also show performance on a three-node cluster with 2.4 GHz Xeon processors running Redhat Linux 9.0 to show the cross-compatibility of FEMPI across different platforms and analyze the performance on a different class of system. The Xeon servers are connected via Gigabit Ethernet.

Figure 2 shows performance of FEMPI's point-to-point communication calls on both the platforms. For one-sided communications using *MPI_Send* and *MPI_Recv*, the maximum throughput reaches about 590 Mbps on the Xeon cluster with 1 Gbps links. This maximum throughput value is comparable to conventional and commonly used MPI implementations such as MPICH [8] or LAM/MPI [10]. For further comparison, we show the throughput achievable with MPICH on the Xeon cluster using *MPI_Send* and *MPI_Recv*. It can be seen that the throughput with MPICH saturates at approximately 430 Mbps. However, MPICH performs better for smaller data sizes because of data buffering. Presently, FEMPI does not implement buffering of data and hence to avoid unfair comparison we do not show MPICH results for any other function call. On the embedded cluster with 100 Mbps links and slower processors, the maximum throughput is approximately 31 Mbps, again comparable to popular MPI implementations. For the *MPI_Sendrecv* function call, throughput is low at around 120 Mbps. In a system with two nodes, data is exchanged between the two processing nodes simultaneously in a sendrecv call leading to possible network congestion and hence lowering the throughput. For systems with more than two nodes, the throughput can increase if the send and receive (that are part of a node) are not with a same node thereby enabling simultaneous communication with different nodes (i.e. data can sent to one node while receiving data from a different node).

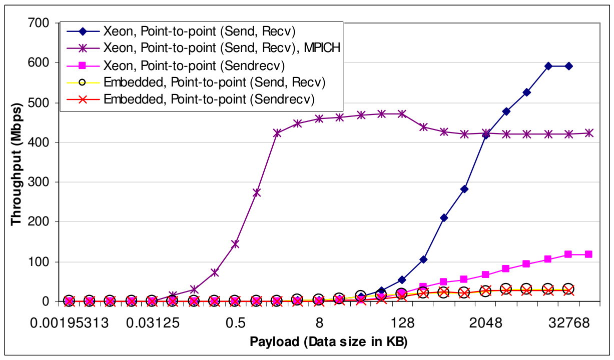
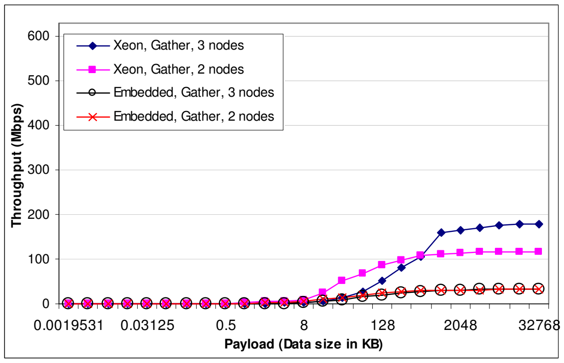


Figure 2. Performance of point-to-point communications

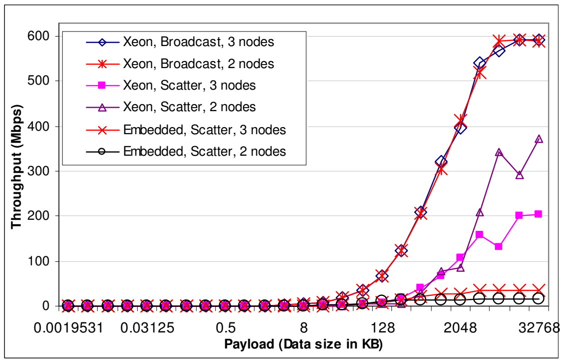
Table 2. Barrier synchronization in FEMPI

No. of nodes	Xeon	Embedded
2	12.9 ms	37.5 ms
3	13.3 ms	57.0 ms

Table 2 gives the timing for synchronization of nodes with FEMPI using the barrier synchronization function *MPI_Barrier* defined by the MPI standard. From the table, it can be seen that the synchronization time is low in FEMPI on the Xeon cluster. As expected, the synchronization time is higher on the embedded system, which can be attributed to the slow processing power and network.



(a) Communication via *MPI_Gather*



(b) Communication via *MPI_Bcast* and *MPI_Scatter*

Figure 3. Performance of collective communications

Figure 3 shows the performance of several collective communication calls in FEMPI. The *MPI_Gather* function is used to gather data at a single root node from all the other nodes. Local data at the root is copied (*memcpy*) from send buffer to the receive buffer while data received from the non-roots are placed directly in the receive buffer. The throughput in Figure 3(a) is low because the root serializes the 'receives' from the clients that are ready to send their data. Once communication has been initiated by the root with a client, all transactions with that client are finished before moving to another client. Figure 3(b) shows that *MPI_Scatter*, where data from one root node is distributed to all the client nodes, performs better than *MPI_Gather*. The better performance of *MPI_Scatter* is due to the fact that the root node does not serialize communication with the clients and data is transferred to all clients that are ready. Figure 3(b) also illustrates the performance of the *MPI_Bcast* collection communication call where data from a root node is broadcast to all the other nodes in the system. The maximum throughput here is around 590 Mbps. It can be seen that performance of scatter is poorer than broadcast. The performance difference is because the root node individually communicates with each client to transfer data (data being unique to each client) in a scatter function whereas with the broadcast the root node publishes to the same data to all the clients using a single transfer. For the same reason, the throughput also does not vary with the number of nodes in a broadcast function.

From the results presented, it can be seen that FEMPI performs reasonably well for both point-to-point and collective communication. In addition, and more importantly, FEMPI provides fault-tolerance and helps applications to successfully complete execution despite faults in the system. Due to space constraints, we only report fault-free performance of limited FEMPI functions calls in this paper.

5 Conclusions and Future Research

In this paper, we presented the design and analyzed basic characteristics of a new fault-tolerant, lightweight, message passing middleware for embedded systems called FEMPI. This fault-tolerant middleware reduces the impact of failures on the system and is particularly important for support of applications in harsh environments with mission-critical requirements. The recovery time of the system is improved allowing unimpeded execution of applications as much as possible. A key focus of the message-passing middleware architecture has been toward tradeoffs between performance and fault-tolerance, with emphasis on the latter but forethought of the former.

Initial results on fault-free performance of FEMPI on an embedded cluster system with PowerPCs and on a traditional Xeon cluster were presented. FEMPI performed significantly well providing a maximum throughput of around 590 Mbps on a Xeon cluster with a 1 Gbps network and 31 Mbps on a cluster of PowerPC-based embedded systems with 100 Mbps network, both roughly comparable to conventional, non-fault-tolerant MPI middleware.

As future work, FEMPI will be tested with several case studies including prominent space applications and benchmarks including multiplication of large matrices, 2DFFT, LU decomposition, with and without injected faults, to provide additional insight on the benefits and costs of fault-tolerance in this environment.

6 References

[1] J. Samson, J. Ramos, I. Troxel, R. Subramaniyan, A. Jacobs, J. Greco, G. Cieslewski, J. Curreri, M. Fischer, E. Grobelny, A. George, V. Aggarwal, M. Patel and R. Some, "High-Performance, Dependable Multiprocessor," *Proc. IEEE/AIAA Aerospace Conference*, Big Sky, MT, March 4-11, 2006.

[2] Message Passing Interface Forum (1994), MPI: a message-passing interface standard, Technical Report CS-94-230, Computer Science Department, University of Tennessee, April 1.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," *SuperComputing 2002*, Baltimore, USA, November 2002.

[4] Agbaria and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations," *In the 8th IEEE International Symposium on High Performance Distributed Computing, IEEE CS Press*, Los Alamitos, California, pp. 167-176, 1999.

[5] "Ensemble project web site", http://dsl.cs.technion.ac.il/projects/Ensemble/ (Accessed: March 1, 2006).

[6] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *In Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawai, pp. 526-531, 1996.

[7] S. Rao, L. Alvisi and H. Vin, "Egida: An Extensible Toolkit for Low-overhead Fault-Tolerance," *In Proceedings of IEEE International Conference on Fault-Tolerant Computing (FTCS)*, pp. 48-55, 1999.

[8] "MPICH project web site", http://www-unix.mcs.anl.gov/mpi/mpich/ (Accessed: March 1, 2006).

[9] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "Mpi-ft: Portable fault tolerance scheme for mpi," *In ParallelProcessing Letters*, *World Scientific Publishing Company*, Vol. 10, No. 4, pp. 371-382, 2000.

[10] "LAM/MPI project web site", http://www.lam-mpi.org/ (Accessed: March 1, 2006).

[11] G. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, Hungary, Vol. 1908, pp. 346-353, 2000.