Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp368-373)

A Fault-Tolerant Parallel Text Searching Technique on a Cluster of Workstations

ODYSSEAS EFREMIDES
University of Hertfordshire, Hatfield, UK,
in collaboration with IST Studies
Dept. of Computer Science
72 Pireos Str., 183 46, Moschato, Athens
GREECE

GEORGE IVANOV
University of Hertfordshire, Hatfield, UK,
in collaboration with IST Studies
Dept. of Computer Science
72 Pireos Str., 183 46, Moschato, Athens
GREECE

Abstract: A fault-tolerant parallel implementation of the well-known Brute Force pattern matching algorithm, using high level checkpointing, is investigated herein. The main objective is to provide reliability: the application returns having completed its computation task correctly. Using the MPICH NT MPI library, which lacks any internal fault tolerance support, the implementation aims at Windows based clusters. The performance of the approach is also investigated experimentally.

Key–Words: Fault tolerance, Parallel pattern matching, High level checkpointing

1 Introduction

Custom made clusters of workstations are more likely to suffer from hardware or network failures compared to standard supercomputers. To ensure the successful completion of the computation process, important information has to be saved on reliable devices, usually at regular time intervals. This data is used to roll back, to the last known correct state, if an error occurs [7] [12]. This process, known as *checkpointing* (journaling), can be implemented at the *system* (low) level or the *application* (high) level.

The fault tolerant adaptation of a relaxed-parallel implementation of the exact Brute Force pattern matching algorithm, using high level checkpointing, is investigated herein. This high level approach requires less storage space and induces less overhead. Furthermore, it is easily applicable to both heterogeneous and homogenous environments [3]. This fault tolerance mechanism is applied on a parallel implementation of the exact string matching problem [5], which is an appealing area in theoretical and practical research, as it is applied in numerous fields ranging from computer science to biology.

2 The Fault Tolerant Approach

2.1 Pattern Matching

Assume a pattern string $P[0 \dots m-1]$ of length m and a large search text $T[0 \dots n-1]$ of length n, with $m \leq n$, where both strings are consisted of characters belonging to a well defined alphabet Σ. Given that, the exact string pattern matching problem is defined as the attempt to discover all occurrences of the pattern P in the text T [5] [15] [14] [16].

A straightforward approach to solve the problem is known as *pattern matching sliding window mechanism* [5] [1] and the simplest realization of it, is the Brute Force pattern matching algorithm. It has no initialization procedure, like the preprocessing of the pattern or the search text, which other algorithms might perform, and starts the scanning phase with no delays. All positions of the search text between 0 and $n - m$ are scanned for occurrences from left to right based on the sliding window. After each attempt to find a match, the window is shifted to the right by one position. The order in which characters are compared within a group is immaterial to the final result. It is a rather slow algorithm and as shown in [5] [1] it has a worst case execution time complexity of $O(m \times n)$ and an expected maximum number of $2n$ character comparisons. Although, the Brute Force algorithm is not an optimal pattern matching algorithm, it was selected due to its simplicity and its computing power demands, since the purpose of this study is to evaluate the performance of the fault tolerant implementation on the cluster. Therefore there is no actual need to be restricted by any specific pattern matching algorithm.

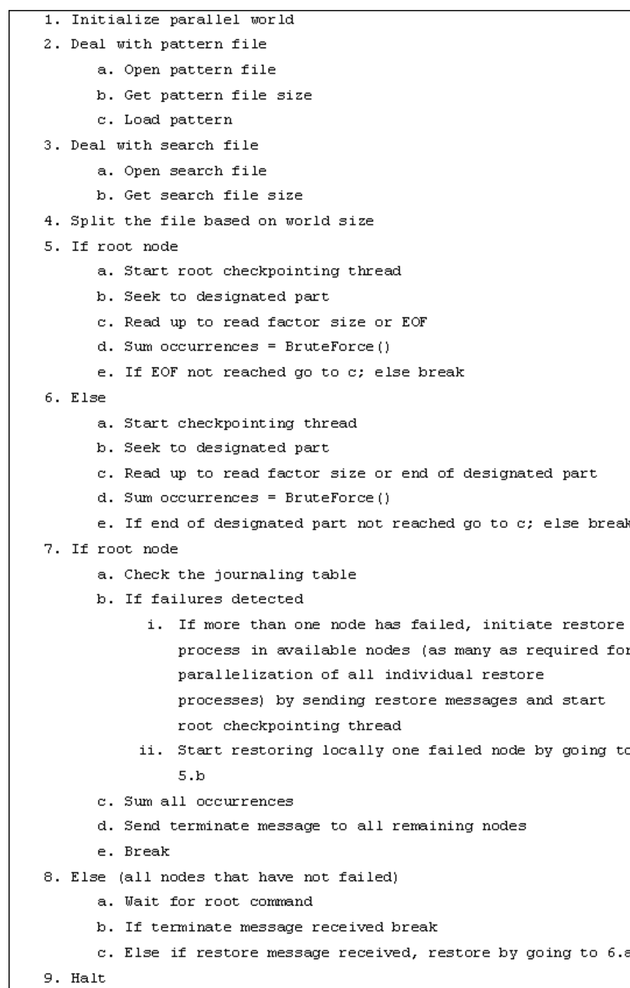
2.2 The Algorithm

The fault tolerance mechanism acts as an extension over the relaxed parallel pattern matching algorithm,

Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp368-373)

which has a rather straightforward implementation: it utilizes available data parallelism over the search files, which are sent to all worker nodes before running the application. Having the search dataset local to all nodes leads to lower execution times as no network traffic is necessary during runtime. The same approach is followed for the distribution of the pattern file. Still, this imposes an overhead to the initialisation phase before running the application. When data becomes available to all workers, each node of the cluster is set to execute the same instructions over a different data stream (or more accurately different part of the data stream), corresponding with the Single Program Multiple Data (SPMD) model over the Multiple Instruction Multiple Data (MIMD) architecture according to Flynn's taxonomy [9].

Fault tolerance functionality is realized using a *thread*, which interacts with the main thread on each node. These threads aim at logging the two important values of the *file pointer* and the *temporarily found occurrences* within each node and then communicating them to the checkpoint server. The thread on the root node builds the checkpoint table, which is used for recovery if needed and for pattern occurrences summation. The way the pattern matching algorithm interacts with the rest of the code should be noted: the function call to Brute Force can be altered with any exact pattern matching algorithm. In particular, steps 5 and 6 exploit the available data parallelism of the algorithm, while steps 5.a, 6.a and 7 are fault tolerance specific. The fault tolerant algorithm is depicted in Figure 1 and explained in detail in the following Sections.



```
1. Initialize parallel world
2. Deal with pattern file
      a. Open pattern file
      b. Get pattern file size
      c. Load pattern
3. Deal with search file
      a. Open search file
      b. Get search file size
4. Split the file based on world size
5. If root node
      a. Start root checkpointing thread
      b. Seek to designated part
      c. Read up to read factor size or EOF
      d. Sum occurrences = BruteForce()
      e. If EOF not reached go to c; else break
6. Else
      a. Start checkpointing thread
      b. Seek to designated part
      c. Read up to read factor size or end of designated part
      d. Sum occurrences = BruteForce()
      e. If end of designated part not reached go to c; else break
7. If root node
      a. Check the journaling table
      b. If failures detected
          i. If more than one node has failed, initiate restore
             process in available nodes (as many as required for
             parallelization of all individual restore
             processes) by sending restore messages and start
             root checkpointing thread
         ii. Start restoring locally one failed node by going to
             5.b
      c. Sum all occurrences
      d. Send terminate message to all remaining nodes
      e. Break
8. Else (all nodes that have not failed)
      a. Wait for root command
      b. If terminate message received break
      c. Else if restore message received, restore by going to 6.a
9. Halt
```

Figure 1: Fault Tolerant Brute Force Text Matching Algorithm

2.2.1 Workload Management

Workload management is static and predetermined based on *MPI world* size and the dataset size and is given by the following formula: $(search_file_size/cluster_size) + 1$. It ignores specific workstation performance characteristics and assigns roughly the same amount of work to all nodes based on their MPI identifiers. Hence, the implementation is targeted at homogeneous configurations for best performance. On heterogeneous systems the execution time is determined by the speed of the slowest node leading to rather unacceptable performance, especially if there are significant differences in the performance capacity of some nodes [16].

Fault tolerance functionality requires that each node is appointed with the task of sending its key values to the root at a predefined interval. The root acts also as a checkpoint server and is responsible for assigning recovery work to nodes. During possible recovery cycles this version may repeatedly add more workload to some nodes, in addition to the workload given at startup, depending on the number of failed and remaining nodes. The amount of work depends on the progress made by the failed nodes to which the work was originally assigned. Again, note that no data is transferred to nodes involved in the recovery process, since a local copy of the entire dataset is already available.

During the computation phase, each workstation only accesses its appointed part, based on a partitioning technique derived from the cluster node numbers assigned by MPI. Segregating the data file introduces a problem in the pattern matching technique: if an occurrence is to appear at the cut point, it will not be found because the first part will be in one node and the last in the following one. Thus, the partitioned data must be overlapped by $pattern_length - 1$ characters [17]. Excluding the last pattern character guarantees that an occurrence will not be counted twice among two different nodes.

Furthermore, data is not only split between workstations but also internally within every node. This provides the ability to handle input files of huge sizes by not being restricted by RAM during the read/load process (signed 64bit addressing support implemented through the WIN32 API). Hence, the internal partitioning to *data packets* of size *read factor* within every node. The read factor has been tested and selected so that it minimally affects read performance. Again there is an overlapping of $pattern_length - 1$ characters to avoid loosing occurrences, which by design does not incur additional I/O overhead.

2.2.2 Checkpoint Mechanism

Concerning the fault tolerance support, checkpointing occurs within every node participating in the scan process, with the exception of the root. It is accomplished independently with no synchronization between nodes, according to their processing capacity and a minimal checkpoint interval threshold. The recovery data is not stored locally but is transmitted to the root node which is responsible for the management of the entire system and acts as a single checkpoint server. The root receives the restore information from all nodes, one at a time and ignores repeated data that may be sent by a slow node. As data from various nodes are received a *checkpoint table* (also called *journal* or *log*) is built and stored in memory. If not all nodes have finished and no data is received at the same time, the fault manager retries for a given amount of time and if there is still no incoming data, the node (or nodes) is considered as failed and thus has to be recovered. That node will also be ignored in all further operations until termination. The retry time should be between 5 and 10 seconds in order to compensate for any network delays.

Obviously, the root node is the single point of failure for the system. If it fails the entire system will also fail. To avoid this limitation the system may be extended with a replica of the root node. This will provide *high availability functionality* for the checkpoint server [6].

Internally within every node the checkpoint mechanism is implemented with the utilization of *threads*. Therefore, the performance of the search process is minimally affected and also remains independent of the fault tolerance mechanism. This is also the reason why the same data may be sent twice from a node: since the two threads are not synchronized at this point, the thread responsible for checkpointing might end up sending the same data if the scanning process advances slower than the checkpoint threshold. As mentioned previously such data is simply ignored by the checkpoint server. Furthermore, the checkpoint process in the root is also implemented using a thread. To either terminate the application or start recovering, the checkpoint thread must first terminate. Then the checkpoint data table is accessed to determine appropriate action. If no failures have occurred the information in the table is utilized to calculate the total number of occurrences of the pattern.

Two vital values are of interest for the pattern matching problem and must be periodically logged to successfully restore the state of a node in case of a failure. They are the *file pointer position* in the search file and the *number of occurrences* currently found. The number of occurrences found depends on the percentage of the search file that has been scanned, thus both values are interrelated and must be synchronized at the time of snapshot. Logging frequency depends on how fast the search file is scanned and what kind of journaling detail is required. Smaller intervals lead to greater accuracy and improved recovery times but induce greater communication and computation overhead and might result in network congestion for large world sizes (as seen in the experimental phase).

2.2.3 The Recovery Procedure

The recovery procedure is controlled by the root node. When the information for one or more nodes in the checkpoint table is found to be incomplete (no finish message received) the recovery process begins. This may be due to any kind of failure either hardware, software, network or their combination. At the first step of the recovery the domain of the search file originally assigned to the failed node is resolved. The last reported progress in the search file is then added to the beginning of that part. At this point overlapping and the temporarily found occurrences are taken under consideration in order not to loose any occurrence. Next is the assignment process of (recovery) tasks to remaining nodes: the checkpoint table is examined and nodes that were last found working are selectively assigned with the recovery work.

The root node can be involved in this process in a twofold manner: it always engages in recovering one node and again acts as the checkpoint server if more than one node is being recovered. Thus, parallelization is achieved when more than one node fail. It is important to note that the recovery process is also fault tolerant and even if nodes that are recovering fail, the work will be eventually completed by other nodes or by the root alone, the last being considered as highly available.

2.2.4 Orchestration & Mapping

The implementation of the pattern matching algorithm is relaxed: no communication occurs between nodes

for processing purposes. However, fault tolerance requirements necessitate the addition of point to point communication of key values between the checkpoint server and all other nodes. Furthermore, in the recovery process the root can dynamically map more work to non failed nodes via network communication. Mapping of tasks to nodes is based on node numbers provided by MPI. Nodes are chosen by searching the checkpoint table and selecting the first available one. Afterwards, details concerning the work to complete are sent to each selected node. If no errors have occurred or when the recovery stage fully completes the root sends a halt message to all nodes, which terminates the application.

3 Experimental Results

3.1 The Cluster Configuration

The cluster that was setup as dedicated [8], has 20 workstations equipped with Intel Pentium 4 2.40GHz CPUs with 512MB of RAM and 20 workstations with Intel Celeron 2.20GHz CPUs with 256MB RAM. Hence, the cluster is either homogeneous or heterogeneous [8] depending on system configuration in each experiment.The interconnection infrastructure of the cluster is a 100Mbit/s Ethernet.

3.2 Development Environment

For implementation purposes, the Argonne National Laboratory MPICH NT Message Passing Interface (MPI) library version 1.2.5 [2] is used, to provide the underlying communication infrastructure between cluster nodes. Although, there exist many implementations of the MPI standard [19], this specific version was chosen mainly due to the fact that it supports Microsoft Windows NT based operating systems, installed on all workstations of the cluster. The application is written in C++ in order to utilize threads under the MPICH NT 1.2.5 message passing library [2] [11] [19]. Note that this library lacks built in failure handling support.

3.3 The Dataset

The application area is genomic research, in which the DNA nucleic acid is scanned for occurrences of sequences. It is currently an active field, with databases that constantly evolve containing a increasing amount of data [14]. Publicly available files from GenBank, a genetic sequences database that belongs to the National Institutes of Health and is part of the International Nucleotide Sequence Database Collaboration [10], were utilized as the search context for this work. Nontheless, dealing with a text database should not be considered a limitation, as only slight modifications are required in order for the implementation to work in any field of pattern matching.

3.4 Results Analysis

The fault tolerant pattern matching implementation is evaluated and compared with the non fault tolerant one when no failures occur, using two search files of size 2GB and 3.5GB respectively and a search pattern of 12 bytes. Note the use of only one pattern, based on the fact that the impact of pattern size in the execution time is rather small as found experimentally and supported by [16].

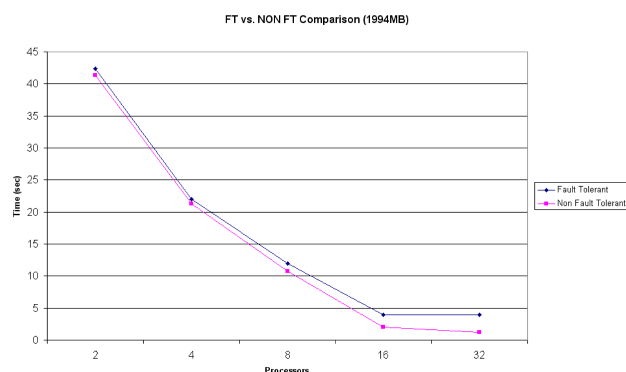


Figure 2: Comparison of Fault Tolerant and Non Fault Tolerant versions using a search file of 1994MB

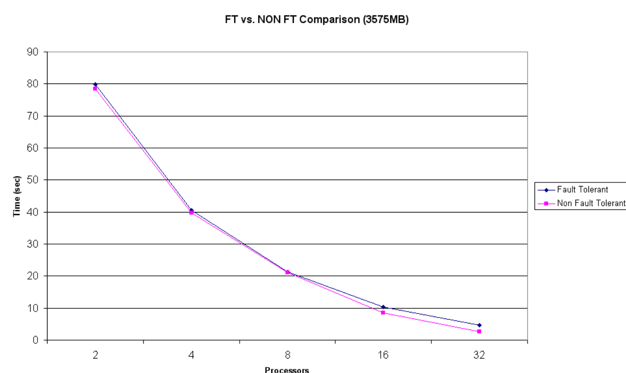


Figure 3: Comparison of Fault Tolerant and Non Fault Tolerant versions using a search file of 3575MB

The *checkpoint interval threshold* was set to a minimum of 1 second, to provide enough journaling detail for each node without inducing excessive overhead. Consequently, a comparison between the two implementations is possible and the overhead of the checkpoint mechanism can be determined. Note that the times reported in Figure 2 and Figure 3 are based on the total execution time which is affected by the

read performance (I/O) of every node and mostly by the *read ahead caching* of the operating system which at some points may present a superlinear performance behaviour. Such results are achieved when the segment of the search file is small enough so that it fits in the read-ahead cache. This occurs in the case of 16 and 32 nodes for the 2GB file and in the case of 32 nodes for the 3.5GB search file. The examination of pure computation times indicated only the normal, expected linear speedup, without superlinearity.

The average fault tolerance mechanism overhead on the total execution time, concerning all search files and all cluster sizes utilized is 1.33 seconds. More than 30 repeated executions were conducted for each test set on the fault tolerant and non fault tolerant versions. The overhead becomes visible as the number of nodes increases, supporting the theory. This intensifies as the search file size decreases, for smaller files are characterized by shorter computation times. This is noticeable for the 2GB file in the case of 32 nodes: while cluster size doubles the performance remains roughly the same as the processing task becomes rather small and is overlapped by the fault tolerance mechanism overhead.

3.5 Fault Tolerance Tests

The fault tolerant behavior of the system was tested by having a consecutively increasing number of nodes fail, starting from one and up to half of the cluster nodes. To further exhibit the notable speed reduction of the recovery process when more than half nodes fail, a relevant test was also conducted. Finally, a node that was recovering the work of a previously failed node also failed in order to show the full extent of the reliability of the system.

Table 1: Fault Tolerance Results (Random Failures)

Nodes That Fail	Total Time (sec)
0	21,251
1	49,288236
2	54,365266
3	65,430271
4	55,20731
5	93,620795
4+1	74,540199

Workstations of the cluster were deliberately made to fail by abruptly cutting off their power supply and by disconnecting them from the network, while computation was in progress, for the purposes of the experimental phase. Obviously, this is not the only type of failure the system can handle: if there is no communication between one or more workers and the root node for the set threshold, then the fault tolerance mechanism will begin the recovery process anyway.

It should be noted that the data that appears in Table 1 may vary even when the same number of nodes fail, because the failure cannot be accurately reproduced, meaning that different nodes can fail at different points of the process and that it is impossible to achieve the same failure type at the same point repeatedly. Nonetheless, the obtained results remain a good indication of the system performance and show that it is reliable and operational as expected.

As the test set, a small cluster size of 8 nodes and the largest search file of 3.5GB were chosen, so that computation would be relatively slow in order to have time to make nodes fail. A pattern of 12 bytes with enough occurrences in the search file was chosen to increase the possibility of an occurrence loss as nodes fail and to clearly show that this does not occur. The number of pattern occurrences in the search file were 3270, all of which were correctly found in all failure cases. The results are reported in Table 1.

Note that in the reported execution times there is an overhead of 10 seconds, which was the chosen *retry time* during which the checkpoint server waits for communication with nodes that have not transmitted their key values and have apparently failed. Taking that under consideration, the time to recover from up to four (4) node failures (half the cluster size) does approximate well the theoretically set time limit. In the case of five (5) node failures and in the last case where one node fails during recovery, the overhead becomes twenty (20) seconds. This happens because the checkpoint server waits twice for a node to respond, first in normal execution and then after the recovery. Thus, results still indicate a good system response time.

4 Conclusion Remarks

A fault tolerant parallel implementation of the Brute Force pattern matching algorithm was investigated herein. Core objective of the implementation is to provide reliable computation results without any user intervention, regardless of the proportion of workstation failures and in an acceptable time frame. The resulting system has a single point of failure which is the root - checkpoint server. Stability for the root can be ensured with additional hardware and software approaches that offer *high availability functionality* on *mission critical systems* [13].

The experimental phase showed an adequate system performance when any number of nodes of the cluster fail (excluding the root node). The scan process successfully completed in all conducted tests, reliably returning all existing occurrences and with-

out any user activity. Furthermore, the execution time in all cases was roughly within expected time limits. Compared to the non fault tolerant version, when no faults occur, the performance of the system is slightly inferior, due to the checkpoint mechanism overhead and communication requirements. Still, the implementation shows excellent speedups as the system scales up to sixteen nodes.

Further work aims in the application of a *load balancing technique* [12] for the checkpoint server which could also noticeably improve performance. Moreover, high availability approaches (such as replication of the checkpoint server) and dynamic workload allocation (for heterogeneous configurations) are under investigation.

References:

[1] A. Aho and M. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Association of Computing Machinery Inc.*, 1975

[2] D. Ashton, W. Gropp and E. Lusk, *Installation and User's Guide to MPICH, a Portable Implementation of MPI Version 1.2.5.* URL: http://www-unix.mcs.anl.gov/mpi/mpich/

[3] G. Bronevetsky, D. Marques, K. Pingali and P. Stodghill, Collective Operations in an Application-level Fault Tolerant MPI System, *Proceedings of the 17th annual international conference on Supercomputing*, San Francisco, CA, USA, 2003, pp. 234–243.

[4] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*, Prentice Hall PTR., 1999

[5] C. Charras and L. Thierry, *Handbook of Exact String Matching Algorithms*, King's College Publications, 2004

[6] G. F. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design*, Second Edition, Addison Wesley, 1994

[7] G. E. Fagg, A. Bukovsky and J. J. Dongarra, *Harness and Fault Tolerant MPI*, Parallel Computing, 2001, 27(11), pp. 1479–1495.

[8] G. F. Pfister, *In Search of Clusters*, Second Edition, Prentice Hall, 1998

[9] M. J. Flynn, Very high-speed computing systems, *Proc. IEEE* 54(12), 1966, pp. 1901–09.

[10] GenBank: National Institutes of Health genetic sequence database
URL: http://www.ncbi.nih.gov/Genbank

[11] W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A high performance, portable implementation of the MPI Message-Passing Interface standard*, Parallel Computing, 22(6), 1996, pp. 789–828.

[12] W. Gropp and E. Lusk, Fault Tolerance in MPI Programs, *Special Issue of the Journal High Performance Computing Applications*, 2002

[13] IEEE Task Force on Cluster Computing. High Availability URL: http://www.ieeetfcc.org/high-availability.html

[14] D. Gusfield, *Algorithms on Strings, Trees and Sequences*, Computer Science and Computational Biology, Cambridge University Press, 1999

[15] P. Michailidis and E. Margaritis, String Matching Algorithms, *Technical Report. Dept. of Applied Informatics, University of Macedonia*, 1999

[16] P. Michailidis and K. Margaritis, Parallel Text Searching Application on a Heterogeneous Cluster of Workstations, *Proceedings of the 2001 International Conference on Parallel Processing Workshops IEEE (ICPPW'01)*, 2001^a, pp. 1530–2016

[17] P. Michailidis and K. Margaritis, String Matching Problem on a Cluster of Personal Computers: Performance Modeling, *In Proceedings of the* 15_{th} *International Conference Systems for Automation of Engineering and Research (SAER'2001), Sofia, Bulgaria*, 2001^b, pp. 76–81.

[18] D. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley, 1997

[19] The Message Passing Interface Forum, MPI: A Message Passing Interface Standard
URL: http://www.mpi-forum.org/

[20] S. Sankaran, J. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman, The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing, *Open Systems Laboratory, Indiana University - Lawrence Berkeley National Laboratory*, 2003

[21] G. Stellner, CoCheck: Checkpointing and Process Migration for MPI, *In 10th Intl. Par. Proc. Symp.*, 1996