Application Level Fault-Tolerance for ScaLAPACK in the NetSolve Environment

Kim Buckner

24 November 1998

Abstract

This paper is a technical report documenting implementation and results of providing for application level fault-tolerance for ScaLAPACK performing a non-trivial computation. NetSolve was used to provide an environment for the stand-alone use of the ScaLAPACK. This is Technical Report UT-CS-98-409.

1 Introduction

Client-server applications are not new nor are parallel computation grids. Applications combining both client-server and parallel computational grids are becoming more popular and fault-tolerance for these applications has not yet been fully explored. This combination should provide more insight into problems involving fault-tolerance.

We chose to investigate the client-server application NetSolve [1] due to its ease of use and ready availability. The parallel computation grid portion was provided by ScaLAPACK [2], a parallel linear algebra package. Both of these are available from NETLIB and are compatible with many architectures. Our goal in this project was to determine the feasibility of providing fault-tolerance that is transparent to the client as well as well as user.

1.1 NetSolve

NetSolve basically consists of an agent which monitors a set of computational resources (servers). Each server informs the agent of the types of problems it can solve. The client requests from the agent a server to solve a problem and the agent returns a listing of the servers available in the order from 'best' to 'worst'. The client directly contacts the first server on the list and waits until that server solves the problem and returns the result or until the server indicates a failure. Upon a failure the client retries that server or goes to the next on the list if it already has seen a failure from the first server. This continues until the problem is either solved or no more servers are available at which time the client reports failure to the user.

The NetSolve client is actually a set of library functions which are accessed by the user through one simple function call. The user needs to know very little about the NetSolve system other than how to link in the client code at compilation.

The try/retry scenario is unseen by the user. The agent does not directly monitor the servers but relies on periodic 'pinging' and reports from the clients and servers. This provides a limited form of fault-tolerance. It does not address the situation where there may only be one server capable of providing the particular service requested, nor does it address the situation in which the service is particularly long-running. It is this last case, the long-running parallel computation which we chose to investigate.

Fault-tolerance with checkpointing requires the integration of a number of decisions including; when to take a checkpoint (interval), where in the code to take the checkpoint, where to store the checkpoint, how to detect failure and how to restart the process. The simple method chosen by the NetSolve developers can serve well in many cases but the implementation of fault-tolerance using checkpointing can be more widely applied. Additionally, the design process can provide unexpected insights into what is really needed to solve a given class of problems.

1.2 ScaLAPACK

ScaLAPACK is just such a class of problems. ScaLAPACK is the parallel implementation of the LAPACK [3] suite of linear algebra routines. It can use PVM or MPI as the underlying communication package and uses BLACS (Basic Linear Algebra Communication Subroutines) [4] on top of these packages.

As our concept was to provide transparent fault-tolerance for a non-trivial parallel application that was widely used we concentrated on ScaLAPACK's function pdgesv(), a parallel LU decomposition and solve. The ScaLAPACK code was modified to provide the application level checkpointing and the NetSolve server was modified to provide the monitoring for fault tolerance.

PVM provides a good interface for the purposes of fault-tolerance. With PVM, the condition of machines on the grid can be queried at almost anytime and the computation can be monitored externally. MPI, even though it is not as 'friendly' as PVM, is becoming the de facto standard for parallel communications. It is being implemented, in various versions, by more and more vendors and so was chosen as more of a real-world test.

1.3 Organization

The remainder of this paper is organized as follows: Section 2 details the modifications to NetSolve and ScaLAPACK to provide the fault-tolerance; Section 3 discusses the tests and results; Section 4 provides conclusions and lessons learned.

2 Implementation

The implementation of fault-tolerance for this research included combining several methods. Multiple checkpoints are stored on separate workstations. *Live* data is tracked and only that part of the process space is checkpointed. Application code is instrumented by hand. Failures are detected by checking the contents of status file. Recovery of the process is by restarting the original routine with an argument which indicates that it is to restore from checkpoint data. More specific information is contained in the subsequent sections.

2.1 Checkpointing

First, the ScaLAPACK routine was modified to provide checkpointing. This consisted of function calls to register and unregister data locations and sizes and function calls to actually checkpoint. These functions were inserted into sections of code that performed significant computation as determined by data collected from uninstrumented code. The ScaLAPACK routines are normally used by calling a *driver* function such as pdgesv(). These driver functions do little actual calculation but only serve to call the working functions in the appropriate order. This necessitated placing some type of information into the checkpoint files that specified which sub-routine was running when the checkpoint was taken. It does allow the data registry to be per sub-routine, so that only live data is saved to the checkpoint file. That is each sub-routine registers and unregisters data as it is needed so that temporary storage is only checkpointed in the using functions.

The checkpoint function first checks the time since the previous checkpoint and if it exceeds the set interval (compiled-in for the tests), the function then forces a checkpoint. As all processors need to participate in each checkpoint, the checkpoint function also serves as a synchronization point for the computation. Checkpoints are saved to a single processor but due to the potential size of the checkpoints in the final tests, it was decided to save each checkpoint on a separate processor in a round-robin fashion, that means that checkpoint one is saved on processor (0,1), checkpoint two on processor (0,2), etc. . Because the input data was written to files on the initial processor, the round-robin started with the second processor so that the original data could serve as the initial checkpoint without having to actually checkpoint the running computation.

Checkpoints and input/output data files were stored in scratch space on the local disks of the workstations. The reason for this was the size of the checkpoints which precluded writing them to the file system (NFS). Each checkpoint in the final tests was 763 megabytes. The input/output data files also totaled 763 megabytes. The round-robin system was adopted for two reasons. The first is necessity. The available disk partition was 1.2 gigabytes and two checkpoints would not fit on a single processor. If the checkpoint was simply overwritten a failure during checkpointing would mean the loss of all data. The second reason is, distributing the checkpoint results in more fault tolerance. It helps guard against a single workstation failure resulting in loss of the entire computation.

2.2 Checkpoint Files

Each checkpoint really consists of two files. One contains information on the computation such as number of processors, name of service and input files. This file also contains the IP addresses of the participating processors along with their position in the grid and the grid position of the processor which took the last checkpoint. The reason for these last items is there is no guarantee exactly which order any particular process will check in when started by MPI.

The second checkpoint file contains the actual data. Data distribution for ScaLAPACK is based on a 2D block-cyclic format and can have a significant amount of overhead. The idea of having to absorb this overhead was not appealing and so it was decided to store checkpoints in a per machine fashion. That means the checkpointing process receives the data from the grid processes in sequential order and in large blocks and writes all the data for a single process before proceeding to the next. This reduces the overhead for gathering a checkpoint to less than one-fifth of that for the 'normal' data distribution. The grid position of the process and the size of the data for that process is written along with the actual data. This means that the restore function has a very easy task as well, simply read a size and then read and send large blocks of data.

2.3 Recovery

MPI has no concept of fault tolerance and hence no to way to cleanly monitor its parallel grid operations other than continually sending keep-alive messages. BLACS further complicates this and there is the potential for the entire computation to 'hang' forever if a processor fails during blocking communications. Because of this we were forced to rely upon a status file created by the actual service routine. The server spawns the service routine via a fork-and-wait and when the child terminates, the service routine validates the status of the computation by checking the contents of the status file. If the computation was not completed (the status file is empty/does not exist) or had errors it is restarted.

In the case of a restart, the restore function must locate the checkpoint and, if multiple checkpoints were taken, determine which is the most current. To do this, each processor checks to see if it has a checkpoint and which grid position took the checkpoint, then a simple comparison determines which is the most current. That processor which holds the most current checkpoint then distributes it and the computation is restarted. Unlike many single processor checkpointing schemes, this one only saves data modified by the process which is not easily reproduceable instead of the entire data/stack segments. When the server restarts the service routine, the service routine executes up to the point of the most current checkpoint via the driver routine. On restore, the driver has been modified to skip all functions that effect the computation which are completely included in the checkpoint up to the function in effect when the checkpoint was taken. This ensures that the correct memory is registered so that the restore function, called from the function that checkpointed, only restores 'live' data. Of course this assumes that the computation is completely deterministic.

3 Results

Final testing of our concept was performed in the Departments Gemini lab consisting of twelve Sun workstations, one Sun Ultra 2 Model 2170, and eleven Sun Ultra Enterprise 2 Model 2170s. Each workstation has two 167-MHz UltraSPARC-1 processors, 256-Mbytes memory, 10/100 Mbps Ethernet interface, two 2.1-Gbyte internal fast/wide SCSI-2 disks, and they are running Sun Solaris 2.5.1. For our tests we used the 100 Mbps Ethernet vice the available 155 Mbps ATM on the basis that the Ethernet connection was more similar to what might be used in a 'real' application. Only one process was spawned on each workstation even though each has two processors. This ensured that there was no memory conflict/contention. The lab was not partitioned of from the rest of the department's network but logins were restricted to one individual and all other jobs not owned by root were terminated.

As previously indicated we chose to use the ScaLAPACK driver pdgesv(), an LU decomposition and solve of a system $Ax = b$. For these tests, A is a 10,000 by 10,000 matrix of double precision numbers and b is a 10000 by 1 vector of doubles. Primary tests were conducted using a 2-by-4 processor grid. The NetSolve agent and the client process were run on workstation 1, the Ultra 2, where the input data had been prepositioned on the local disk. The server and the processor grid used the Enterprise 2 machines. The matrix and the vector are distributed by the service routine in a 2-D block-cyclic fashion.

The results of the 2-by-4 grid tests are in Table 3. The first entry, "Unmodified Code", is for the original NetSolve and ScaLAPACK code without the checkpointing modifications. The "No Checkpoints" entry is for the modified code with the checkpoint interval (compiled-in) set longer than the possible running time. "With Checkpoints" is time with two checkpoints taken.

The last two lines indicate times with failures. The "Failed Before Checkpoint" shows the time if a failure is induced before the checkpoint is taken and the process is then restarted and allowed to finish. The last entry shows the case where a failure is induced after the checkpoint and the process restarted and restored from the checkpoint. The checkpoint interval is set to 45 minutes for those cases where checkpoints are taken. The failures before checkpointing were induced at approximately 30 minutes into the computation and the failures after checkpoints occurred at approximately one hour.

"Total" is the amount of time required to receive the data from the client, start and run the service routine (including checkpoint and recovery) gather the result and send the data back to the client. "Ckpt" is the amount of time for the process to gather the checkpoint and store it to disk. "Rest" is the time for the process to read the checkpoint from the disk and distribute the data to the grid. "Comp" is the time for the actual computation excluding the time to start the service routine and communicate data with the client.

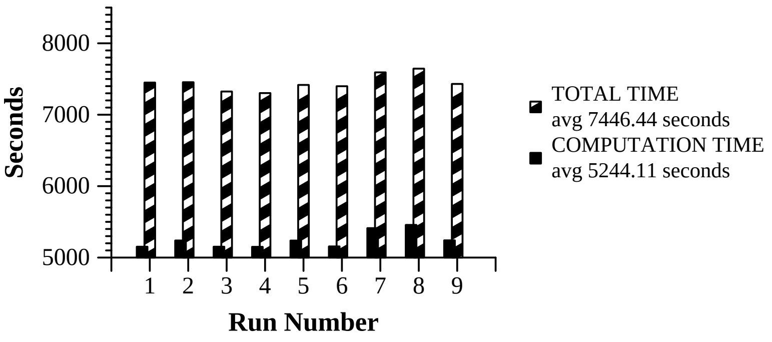
The graphs which follow detail the test performed in the Gemini lab with

Test	Total	Comp	Ckpt	Rest
Unmodified Code	7446.44	5244.11	0.00	0.00
No Checkpoints	7819.67	5542.00	0.00	0.00
With Checkpoints	8171.00	5834.33	146.17	0.00
Failed Before Checkpoint	10259.67	6853.67	144.33	0.00
Failed After Checkpoint	9817.33	7381.33	145.23	257.07

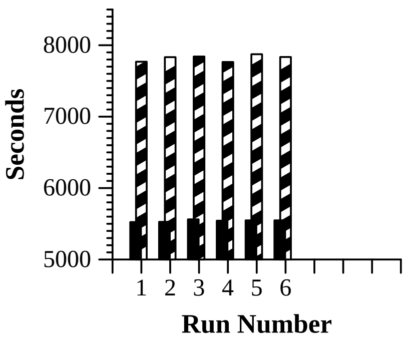
Table 1: Running Times (All times in seconds)

the lab reserved. In all cases the tests are of the same problem (pdgesv()) with input matrix **A** and vector **b** as previously specified.

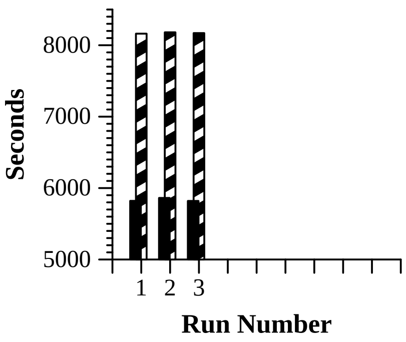
This first set of graphs shows the comparison between total and computation times for the 2-by-4 grid for unmodified code, modified code with no checkpoints taken, and modified code with checkpointing but no failures.



Unmodified Code

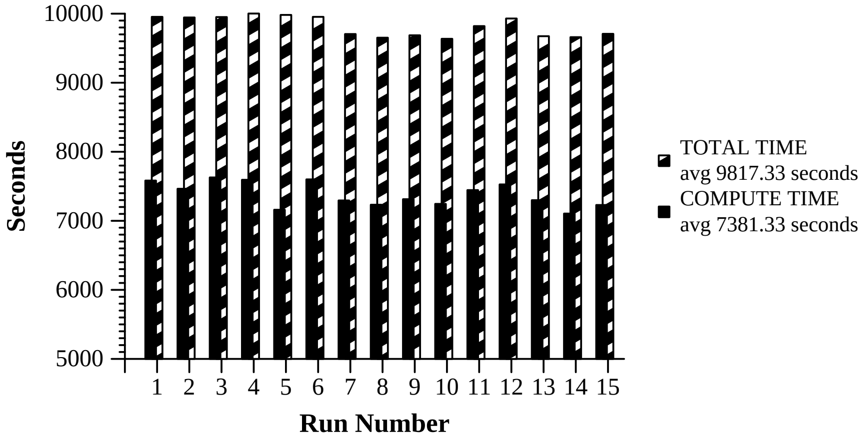


No Checkpoints

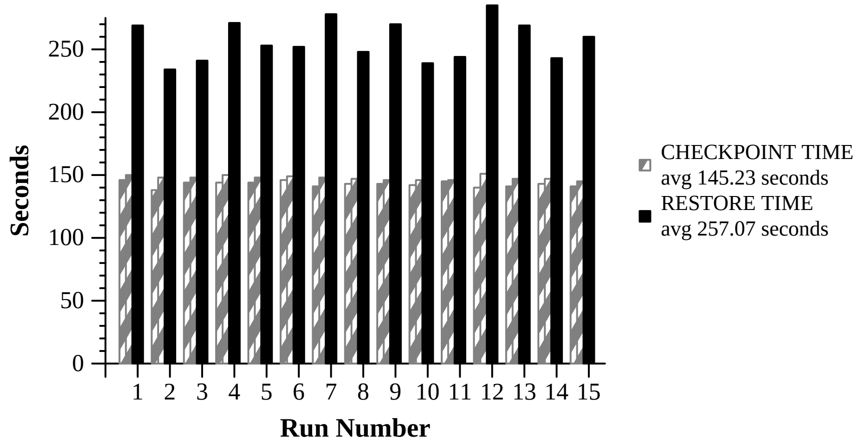


With Checkpoints

Here we see the times for the 2-by-4 grid in the presence of failures which occur approximately 45 minutes into the computation.

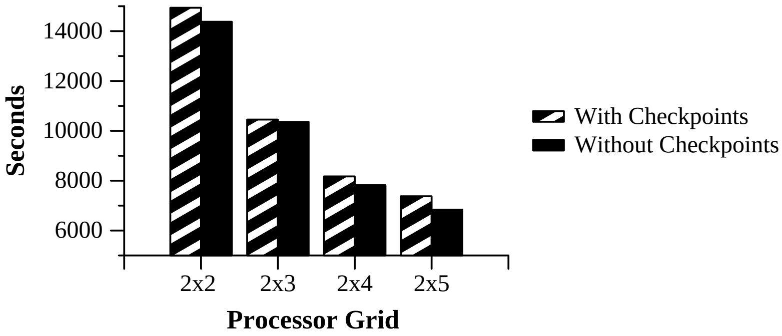


Processor grid 2 x 4
Times with Checkpoint/Fail/Restore

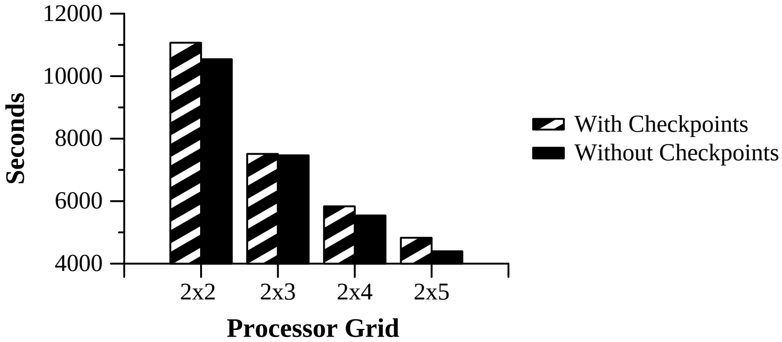


Processor grid 2 x 4
Checkpoint and Restore Times

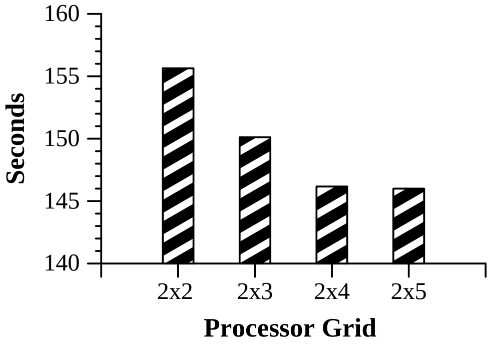
After having performed the tests for the 2-by-4 processor grid, we decided to run the code for different grid sizes. The intent was to gather information on checkpoint time and running time as the grid size changes. Given the method of distributing the data for ScaLAPACK, as the number of processors changes, the size of data per processor changes proportionally. The question then was does the running time scale the same way. These show the comparison between total time, computation time and checkpoint time as processor grid size changes.



Total Time

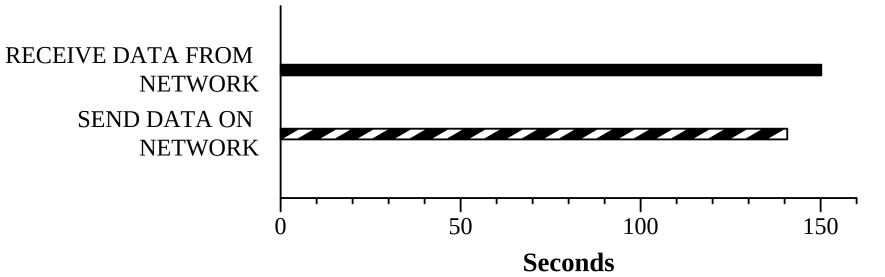


Computation Time

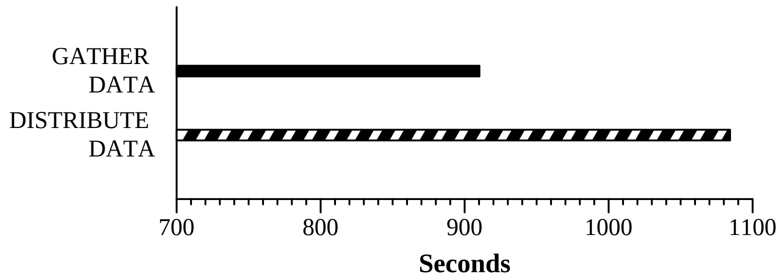


Checkpoint Time

Because communication overhead is significant we show the communication times to 1) send the input/output data across the network from one local disk to another within the Gemini lab and 2) the time to distribute (2-D block cyclic) the input data across the process grid and to gather back the answer.



Network Communication Time



Data Redistribution Time

4 Conclusions

Providing application level fault-tolerance is a very reasonable approach to the problem of long running parallel applications. Granted not every application will fit so neatly into the available memory and processor resources but given that many users of NetSolve have only large applications (as opposed to huge) and use standard libraries, this is a very simple method to provide fault-tolerance of a higher order than simply restarting the application from the beginning complete with resending the original data.

The modifications to the application required to perform this fault-tolerance are minimal and could certainly be incorporated in a library that could be used to instrument other code. The modifications to NetSolve are also minimal and given the modular implementation of NetSolve could be included in a release without having to modify all of the NetSolve code.

The idea of coordinated checkpointing seems to be wasteful of computation time but in this case as each process completes sending its current state to the checkpointing process, it is free to continue computation. The same applies to recovery. After a particular process has received its share of the data it may proceed. As is shown the two checkpoints are only 5% of the total computation time and only 3.6% of the overall total in the 2-by-4 grid.

As expected, the cost of restarting the computation from the beginning exceeds that of restarting from a checkpoint. Restarting from the beginning, NetSolve's basic fault-tolerance, could be considered equivalent to a failure before having taken a checkpoint. Comparisons of the data in Table 3 shows that checkpoint-restore is faster. The seeming anomaly in the the computation time and the total time for these two cases is explained by the fact that the failures before checkpoints occurred after 25 minutes of computation. Total time for the computation without failures is approximately 90 minutes giving 115 minutes total computation time. For the failure with recovery case the computation was allowed to run approximately 70 minutes before failure and then took an additional 53 minutes after recovery for a total of 123 minutes. If the process had been allowed to run without a checkpoint for 70 minutes and then had a failure induced the total time for the computation would have been on the order of 210 minutes. The fact that the total time for the process which failed before checkpoint is still longer than the one that failed after the checkpoint is explained by the very costly 2-d block-cyclic data distribution. The algorithm for that is under examination for improvement but much may depend on portability issues.

One other item that this research brought into focus was the need to preposition data. Initially we tried to store the input data in the NFS file system. The actual disk the data was on resided at the opposite end of the building on another sub-net. When trying to send the data from the client to server, reading from the files and sending across the network, the server for the data disk experienced periods in which it was unavailable due to the increased traffic. This also occurred when trying to write a checkpoint to the same disk, the writes would fail due to unavailability of the server. We considered trying to store the checkpoint locally in the Gemini lab then moving it with a separate process, however the computation had large compute-intensive segments allowing little time for the secondary process to run and due to this and NFS latency, the checkpoint was not actually moved until AFTER the computation completed, at which time the server again became overloaded.

References

[1] Casanova, Henri and Jack Dongarra. "NetSolve: A Network Server for Solving Computation Science Problems" *The International Journal of Supercomputer Applications and High Performance Computing*, Volume 11, Number 3, pp 212-223, Fall 1997.

[2] Choi, J. , J. Demmel, I. Dhillon, J. Dongarra, et.al. LAPACK Working Note 95, ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. Technical Report UT-CS-283-95, University of Tennessee, 1995.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, .et.al. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. Technical Report UT-CS-90-105, University of Tennessee, 1990.

[4] Dongarra, Jack J. and R. Clint Whaley. LAPACK Working Note 94, A Users Guide to the BLACS v1.1. Technical Report UT-CS-95-281, University of Tennessee, 1995.