A prototype of communication structure enabling some degree of fault tolerance for the DOUG parallel FE solver

Konstantin Skaburskas, Eero Vainikko
Institute of Technology, University of Tartu
Vanemuise 21, 51014 Tartu, Estonia
{konstan, eero}@ut.ee

Abstract.

Current HPC systems are increasing in size with higher potential for individual node failure. The need for using fault tolerant implementations of long-running parallel computational programs and packages rises. An overview of different methods to achieve fault tolerance with MPI programs is presented. We have devised and implemented a prototype of communication structure enabling some degree of fault tolerance for the parallel FE solver DOUG. The DOUG package is a black box parallel iterative solver for large sparse linear systems of equations arising from finite element discretization of elliptic partial differential equations. To archive certain fault-tolerant behaviour we exploit a dynamic process model of Message Passing Interface (MPI) present in LAM-MPI implementation of MPI-2 standard. To save computed intermediate data (needed during recovery) we use application-level user-directed checkpointing based on MPI-2 non-blocking File I/Os.

Keywords: MPI, fault tolerance, HPC, parallel linear system solver

1 Introduction

With the increasing size of high performance computing (HPC) systems and development of the Grid computing environments one or several computational processes are more likely to become unavailable during parallel computations. While the distributed systems community has been studying thoroughly the possibility of software systems implementation that can tolerate hardware failures the parallel computing community has generally ignored this problem. The reason is that until recently, most parallel computing has been done on relatively reliable machines. However, new trends in HPC, such as the popularity of custom-assembled clusters, increases the probability of hardware failures. Also, the running time of many computational science applications is nowadays significantly greater than the mean-time-between-failures (MTBF) of the hardware they run on [3]. Therefore, one has to admit that fault tolerance (FT) is becoming a critical issue on HPC platforms and the need for fault-tolerant implementations of long-running parallel computational packages is extremely important.

Message Passing Interface (MPI) is a straightforward and effective way for writing parallel programs for parallel and distributed computers. There are a number of MPI Standard implementations [14]. However, MPI Standard does not define parallel process fault recovery procedures and, as a consequence, developers of standard-conforming MPI implementations do not embed such features into their code. Currently, from the relationship between MPI Standard, its implementations and parallel applications it turns out that FT must mainly be a property of parallel applications themselves with some support from MPI implementation.

In addition to message passing routines, MPI offers a programmer some flexibility by specifying master/slave and client/server models for dynamic process management. In this paper we present an attempt to achieve fault-tolerant behaviour of parallel programs by using dynamic process model of MPI. The implementation is mainly designed for the purpose of the computational package DOUG [5, 8 and 10]. We build the fault tolerant system from standard

widely available tools. We expect that given approach can be utilised also for computations on the Grid in the future.

This paper is organized as follows. In Section 2, a brief description of the DOUG parallel FE solver and its communication pattern is given. Section 3 is an overview of existing systems and approaches for writing fault tolerant MPI programs - approaches that are possible and appropriate within the context of MPI. This section shows also why problems arise within the context of static process model of MPI. In our approach conventional computation methods are amended with some elements of distributed computing methodology, which is described in Section 4. In the final Section 5, we conclude with some pros and contras of the chosen approach and give some guidelines for the future work.

2 The parallel FE solver DOUG

2.1 Package description

The DOUG (Domain Decomposition on Unstructured Grids) is a black box parallel iterative solver for large sparse linear systems of equations arising from finite element discretization of elliptic partial differential equations. Written in Fortran 77, it uses MPI for interprocess communication. Used in conjunction with any finite element discretization code, the DOUG can solve the resulting linear systems using a Krylov subspace iterative method powered by a range of domain decomposition preconditioners. The code is designed to run efficiently in parallel on virtually any machine that supports MPI. Using graph partitioning software all the operations needed in any iterative solver (matrix-vector multiplication, dot-product) are parallelised. Additive Schwarz preconditioners can be automatically constructed by the DOUG with only minimal input. In the first release of the code a full Additive Schwarz preconditioner with automatically generated coarse grid was provided in both 2D and 3D.

We give here only a brief description of the DOUG features, mainly concentrating on communication. A more thorough overview of the package and its recent developments can be found in [10]. Suppose, we have a partial differential equation (PDE) to solve which describes some physics of a phenomenon we are interested in on a given domain. We discretise the domain by generating a set of finite elements (FEs) on a chosen set of grid nodes where we want to obtain the solution of the PDE.

The automatic parallelisation is achieved as follows: the set of FEs is divided into the desired number of subsets using graph-partitioning software METIS [9]. This produces a number of subdomains with some overlapping nodes between neighbouring regions. After distributing the subdomains to different processors, the overlap is the reason for data dependencies which results in communication between neighbour processes. This communication pattern arises in the following building-blocks of a Krylov subspace iteration methods:

-- after each matrix-vector operation;
-- after each application of Additive Schwarz preconditioner.

In the Additive Schwarz preconditioner on each subdomain an exact or inexact solver is applied. The results from the neighbouring subdomains are added on the overlapping nodes. In addition, iterative methods need the dot-product operation which in parallel case gives rise to the need for all-to-all (global) communication pattern.

The DOUG is using two-level preconditioning technique -- a coarser grid is automatically built upon a fine grid. Coarse grid problem is solved simultaneously with the fine grid subdomain solution. This is achieved by appropriately defining interpolation and restriction operators and coarse grid matrix. Communication between the coarse grid solver and the subdomain processes is needed. Fine and coarse grid solutions are added up during each preconditioning step.

2.2 Structure of the parallel implementation and communication contexts

The current version of the DOUG is using the master/worker logical structure. Master has a global communication context with workers and they in turn have their own one as well. These two communication contexts are independent. Master is responsible for

-- reading in the mesh and initial data;
-- graph partitioning;
-- distribution of work among workers;
-- generation of coarse grid and its calculation (two level Additive Schwarz method);
-- following the convergence along with the slaves;
-- gathering the final solution from the slaves and the output.

Workers are responsible for subdomain solutions and collective computations (like performing Ax-operation and dot-products).

Recently, the DOUG has been used in eigenvalue computations for stability assessment of fluid flow problems [6] which need a number of consecutive solutions of large linear systems with block structure. Such computations can take up to a week on a cluster of workstations and we have decided to provide package with ability to recover from a class of hardware and system faults. Before presenting our approach we start with an overview of different methods to achieve fault-tolerant behaviour of MPI parallel programs.

3 FT in MPI programs

Approaches for the FT in MPI programs originate from four major levels of survivability of parallel programs, defined in [7]:

-- the highest level of survival is that the MPI implementation automatically recovers from some set of faults and the MPI program, regardless of its structure and continues without any significant change to its behaviour;
-- a second level of survival is that the program is notified of a problem and it is prepared to take corrective action;
-- a third level of survival is that certain MPI operations, although not all, become invalid;
-- a fourth level of survival is that a program can be aborted and be restarted from a checkpoint (here the states of all processes are saved outside the processes themselves, typically on a disk).

The approaches given above can be combined appropriately.

Following from statements above four major approaches of fault tolerance in MPI programs arise:

1. modification of MPI semantics;
2. restructuring MPI programs (inter-communicator and MPI-2 dynamic process model);
3. extending the MPI specification;
4. checkpointing (with restarts) and message-logging.

Combinations of the above are possible as well.

3.1 Communicator

Communicator is a fundamental concept of MPI. This is a distributed object that supports both collective and point-2-point communications. It joins processes in distributed environment and creates a context for communication between them. E.g. MPI_COMM_WORLD is the default communicator which is given to parallel program with its initialisation, comprising all processes running within the given parallel program. Current semantics of MPI indicate that a failure of MPI processes or communication causes all communicators associated with them to become invalid.

In standard parallel program using MPI processes share a *communicator(s)* (communication context), which is given to the parallel job during its initialisation. Processes within communicator are numbered, each having its own rank; process can belong to several different communicators. Processes can not be physically removed from this communicator (shrink operation is not defined). It is possible to add new processes to a communicator (spawn new processes with successive integration into an existing communicator). The extension of the number of processes is a global operation within communicator being extended, meaning that all processes belonging to the extending communicator must participate in this operation. According to the standard, all kind of global communications within any kind of communicator are defined if and only if the communicator has no "holes" in it, i.e. all processes are alive. This causes a problem in case of a failure of any number of computational nodes. In this case the communication context for global operations is invalidated (as well as the point-2-point communications with died ranks (processes) within communicator), hence it is impossible to spawn new processes within this communicator any longer. It leads to inability to recreate lost computational nodes and integrate them into previously built communication structure. There is no simple nor straightforward way how to address this problem. Still some elaborate approaches and tricks can be applied.

3.2 Modifying MPI semantics

One way to handle hardware and software faults would be to have an MPI implementation with built-in fault tolerance. In such implementation MPI objects (like communicator) could be given more states (in case of communicator, more than the two states currently in the MPI Standard – valid and invalid). Moreover, semantics of some MPI functions need to be slightly modified to achieve desirable results.

For example, in the case of FT-MPI [4] implementation failures do not cause instant application termination if there is either any hardware, application or system failure. FT-MPI provides user with 5 modes of recovery by extending the states of communicator. They affect the size (extent) of the communicators and ordering of the processes in it:

- ABORT: just do as other implementations;
- BLANK: leave holes but make sure collectives do the right thing afterwards;
- SHRINK: re-order processes to make a contiguous communicator (some ranks change);
- REBUILD: re-spawn lost processes and add them to MPI_COMM_WORLD;
- REBUILD_ALL: same as REBUILD except rebuilds all communicators, groups and resets all key values etc.

Then recovery is done via slightly changed semantics of the MPI functions.

If a failure has been detected, the implementation goes through some certain recovery steps and after that you achieve what you asked for - either re-ordered contiguous set of processes (if SHRINK mode was chosen) or re-spawned processes with preserved ranks in which case the implementation does the trick with filling in the hole(s) in the communicator. Here is a performance cost at recover time mostly, because there is no need to restart application and the parallel application continues to run without restarting.

However, one of the drawbacks of using Standard non-conforming implementations is that then coding assumes

that particular implementation is installed on the cluster, which may not be the case on Grid environment.

While this is a very intriguing approach, the practice of using MPI Standard implementations, which tend to change the MPI semantics is not widely accepted in parallel community and sometimes is treated as inappropriate one. However, MPI Standard is in constant development and it is possible, of cause, that in some future releases some of the FT-MPI key ideas may be adopted.

As it will be shown in Sections 3.3, 3.4 and 4, rebuilding or recreating structure of an MPI job can be achieved by using the MPI dynamic process model coupled with association of error handlers with communicators.

3.3 Error handlers for failing communicators

MPI Specification gives possibility to associate error handlers to communicators. The built-in error handlers are MPI_ERRORS_ARE_FATAL and MPI_ERRORS_RETURN. The statement of MPI_ERRORS_ARE_FATAL specifies that if an MPI function returns unsuccessfully then all the processes in the communicator will abort, which is the default behaviour in case of any fault. The statement of MPI_ERRORS_RETURN specifies that the MPI functions will attempt to return an error code. From the programmer's point of view this implies checking of (almost) every MPI call. Also, programmer is allowed to attach his or her own error handler to communicator, by defining:

- MPI_COMM_CREATE_ERRHANDLER(my_recovery_funct, &errh)
- MPI_COMM_SET_ERRHANDLER(MPI_COMM_WORLD, errh).

In the case of user-defined error handler, all application recovery operations can occur in the handler written by the programmer, which in some cases can be rather sophisticated. Hence, in this case every MPI call does not need to be checked like with MPI_ERRORS_RETURN.

From the above it is clear that FT is a property of an MPI program coupled with an MPI implementation. Implementation detects failures during communications, and then whether returns an error code from an MPI function or calls programmer's error handler. At this stage some particular action with respect to error handling on the level of the program can be taken. However, all that can be done here is just saving intermediate data and gracefully exiting (assuming that there is a dependence on data between processes). Right from this moment communicator has a "hole" and performing any collective communication is impossible, because communication context from now on is considered to be invalid.

3.4 Dynamic model of MPI-2 and inter-communicator

MPI-2 Standard however offers some more flexibility. It allows spawning new processes with the following commands:

- MPI_COMM_SPAWN()
- MPI_COMM_SPAWN_MULTIPLE()

and connecting separately running processes using client/server approach:

- MPI_PUBLISH_NAME()/MPI_LOOKUP_NAME()
- MPI_COMM_CONNECT()/MPI_COMM_ACCEPT()

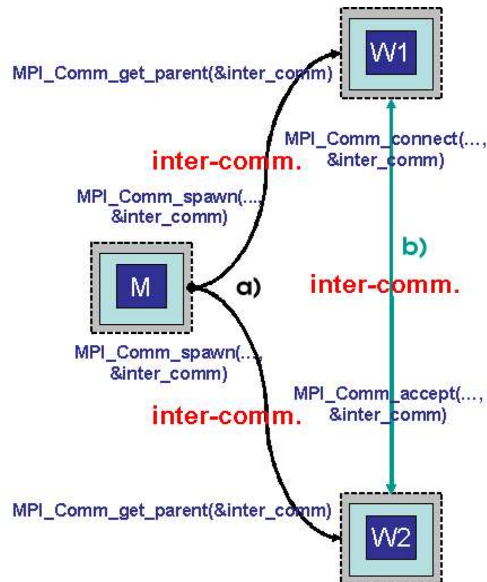

Figure 1. Master/worker and client/server setups; inter-communicators.
a) master (M) spawns two workers (W1, W2); b) workers connect with each
other in client/server fashion. In both cases inter-communicators join corresponding parties.

An MPI process (or group of processes) is able to spawn processes. In every call to MPI_COMM_SPAWN [MULTIPLE]() on master side and supplementary MPI_COMM_GET_PARENT() on a spawned worker side a, so called, *inter-communicator* is returned (Fig. 1). This is the MPI object that corresponds to the both two calling parties. It contains two groups of processes, and communications occur between processes in one group and processes in the other group. In the case of communication via the inter-communicator the failure on one party has no significant effect on the other. The behaviour is similar to in non-MPI client/server programs. The same applies more naturally to MPI client/server setup. Separately spawned processes can find each other via calls to MPI_PUBLISH_NAME(), MPI_LOOKUP_NAME() and connect or accept connections by MPI_COMM_CONNECT()/MPI_COMM_ACCEPT(). In this case inter-communicator is returned as well (Fig. 1).

Appropriate error handlers can be set to the inter-communicators. Then, in case of master/workers setup, if one of the workers dies, implementation calls the error handler on the master side (or other worker side who can signal master about the problem) and then master can perform some recovery actions, like re-spawning died worker and re-newing communication context. Then workers can dynamically reconnect with each other. Hence, the structure of a parallel application can be created and re-created at the parallel programmer level.

3.5 Checkpointing and message-logging

Checkpointing makes programs fault tolerant by periodically saving their state and restoring this state after possible failure. There are 3 different levels of checkpointing:

-- system-level,
-- application-level,
-- user-driven-level.

System- and application-level involuntary checkpointing with successive restarts is a technique which does provide users with so-called transparent FT. System-level checkpointing protocols require all processors to save core-dump

style snapshots of their computations periodically on stable storage; upon failure, all processors resume execution from the last snapshot. Unfortunately, the amount of data saved at each checkpoint can overwhelm the disk storage system, so few high performance machines support or encourage this style of obtaining fault-tolerance [2].

Here are some interesting solutions from application-level involuntary checkpointing. User Transparent, Fault Tolerant, Grid-enabled MPICH - MPICH-GF [15] allows dynamic attachment to user's processes and automatically and involuntary checkpoints them. Another solution is LAM-MPI 7.0 [13] implementation of MPI Standard which offers check-pointable TCP communications by exploiting Berkeley Labs Checkpoint/Restart package [11], thus providing applications running on it with transparent FT. However, the total amount of data saved at check-points can still be too high, while at restarting only a little portion of it is relevant. No doubt, that those and many other very sophisticated approaches have found their users. So, depending on an application one may or may not choose to use application-level involuntary checkpointing (which leads to transparent fault tolerance).

The previous two approaches were automatic checkpointers, while the one developing at Cornell University [12] is a flexible semi-automatic checkpointer tunable by user. It belongs to application-level user-driven check-pointing solutions. This approach needs redesign or introduction of minimal changes to the application to add support for check-pointing. But in contrast to hundred megabytes of information that would be saved during system- or application-level checkpoint, a few megabytes of relevant information is worth of working on the code by the user.

Message-logging is another approach to tackle problem of fault tolerance. This approach has been studied intensively by the distributed systems community, however the overhead of saving or regenerating messages tends to be so overwhelming that the technique is not competitive in practice [2]. This is due to the fact that parallel programs communicate more data more frequently than distributed programs.

4 Fault tolerance and communication structure in the DOUG

We decided to build a fault tolerant system based on widely available standard tools. Our approach is based on standardised dynamic model of MPI and simple user-directed checkpointing by utilizing MPI-2 non-blocking File I/Os. In our implementation we are utilizing some combination of approaches 2 and 4 for building FT MPI programs stated in Section 3, and were more precisely described in Sections 3.3-5.

4.1 Communication structure

In our approach we use the same master/worker logical set-up as the previous version of the DOUG has. However, the whole structure is being changed a little. Master process does not take care of the coarse grid solution. Its task is to manage the whole process of calculations only. We create a separate group of workers (called "coarse solver") to perform this task, thus enabling this part of computations to be done in parallel as well. (For further description see Fig. 2.)

At first, we run one MPI job consisting of one MPI master process. Then we create a master/worker setup by spawning needed number of processes. The workers manage to connect with each other in the client/server fashion (without deadlocks, of cause), and form the needed topology. Appropriate error handlers to each inter-communicator are attached. The same is done for the coarse solver. Workers and the coarse solver communicate via master. Moreover, we create a set of non-blocking receive calls on master to asynchronously serve and manage workers and coarse solver. In case of some failure our own error handler is called by the MPI implementation. According to the recovery protocol dead processes are recreated by master and integrated back into the calculation.

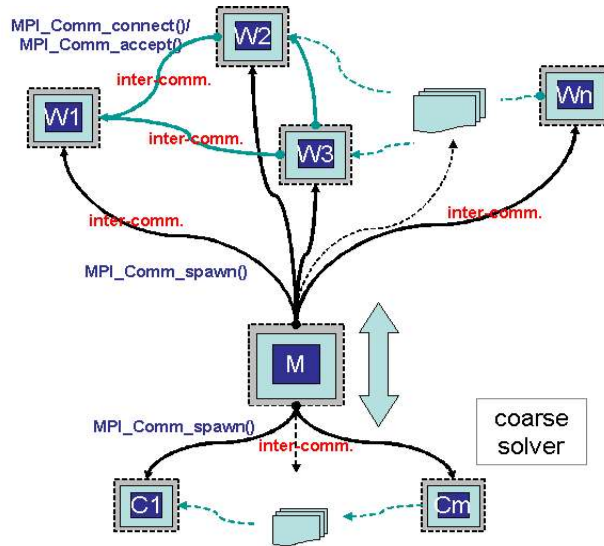

Figure 2. Communication structure of the DOUG package.

In this situation (when we do not have a global communicator upon which standard MPI collective operations can be applied) we are left without any collective operations, like MPI_[ALL]REDUCE(). Therefore, we have to implement them ourselves. Currently system works fine in case of death of one worker, but problems still arise when more workers die.

4.2 Data recovering

On one node after every iteration the only valuable information for us is a solution vector, but not the state of the entire process, which in some cases could be up to Gigabytes. Technically speaking, in case of faults we can recover the full computational state from relatively small amount of data saved at key places in our program. Here we used so called user-directed checkpointing method, where we are responsible for saving computational state periodically, and for restoring this state after failure. This approach is very clear and advisable in our case, since our application is relatively well structured and we can simply insert non-blocking MPI File I/O calls.

5 Conclusions and future work

The main drawback of the approach taken is that it leads to a painful program restructuring and necessity of implementing some global communication operations. However, the communication pattern developed has an interesting property. It naturally creates a possibility for exploiting dynamic process model. E.g., it can be utilized in a case, when due to bad convergence in a sub-domain (residing on a computational node) we would like to split it up into two or more pieces by creating a number of new processes and with successive integration of them into the calculation. This procedure can be performed more easily within the developed communication model rather than within standard static model.

One of the appealing features of implemented fault tolerant communication structure is relatively straightforward application to an asynchronous iterative solution model. Experimental results [1] on systems of linear equations in some cases show advantages for parallel iterative methods without any synchronisation at all. In this case, processes (each having their own part of the problem domain) instead of communication with neighbours after each iteration, communicate asynchronously depending on the need for global convergence. Asynchronous domain decomposition methods look actually most promising for FT implementation.

We have written a prototype of the FT communication model using C. Currently, work is continuing on porting it to FORTRAN95 and integration with computational package is pursued. The main idea is to let the user of the package to decide on compile time what communication model of the parallel process to use - whether standard static MPI process model or dynamic one with or without fault tolerant behaviour. Extensive testing has not been performed yet. So, this would be one of our future works along with qualitative and quantitative comparison of our solution with the other ones (e.g. [2, 4]).

Acknowledgments

This work was supported by the Estonian Science Foundation (ETF) grant No. 5316 and the Estonian Ministry of Education grant No. 018 2565s03.

References

[1] G. M. Baudet, Asynchronous Iterative Methods for Multiprocessors, *J. of the ACM 25*, p. 226-244 (1978).

[2] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, Automated application-level checkpointing of MPI programs. In *Principles and Practices of Parallel Programming,* San Diego, CA, June 2003.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, Collective operations in an application-level fault tolerant MPI system. *In International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23-26, 2003.

[4] G. Fagg, 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: Springer Verlag, 2000), Vol. 1908, 346-353, 2000-01-01.

[5] I.G. Graham, M.J. Hagger, L. Stals, E. Vainikko, The DOUG package. Online at http://www.maths.bath.ac.uk/~parsoft/doug, 2004.

[6] I.G. Graham, A. Spence, E. Vainikko, Parallel iterative methods for Navier-Stokes equations and application to eigenvalue calculation, in: B. Monien, R. Feldman (Eds.), *EuroPar2002 Parallel Processing*, in: *Lecture Notes on Comput. Sci.*, Vol. 2400, Springer, Berlin, 2002.

[7] W. Gropp, E. Lusk, Fault tolerance in MPI programs, *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[8] M.J. Hagger, Automatic Domain Decomposition on Unstructured Grids DOUG, *Adv. Comput. Math.* 9 (1998) 281-310.

[9] G. Karypis, V. Kumar, METIS: Unstructured graph partitioning and sparse matrix ordering system, *Department of Computer Science, University of Minnesota Report*, Minneapolis, August 1995.

[10] E. Vainikko, I.G. Graham, A Parallel Solver for PDE Systems and Application to the Incompressible Navier-Stockes Equations, to appear in *Applied Numerical Mathematics*.

[11] BLCR (Berkeley Labs Checkpoint/Restart) package. Online at
http://www.nersc.gov/research/FTG/checkpoint/, 2004.

[12] Intelligent Software Systems, Cornell University. Online at http://iss.cs.cornell.edu/ft.html, 2004.

[13] LAM-MPI. Online at http://www.lam-mpi.org, 2004.

[14] MPI implementation list. Online at http://www.lam-mpi.org/mpi/implementations/, 2004.

[15] User Transparent, Fault Tolerant, Grid-enabled MPICH - MPICH-GF. Online at :
http://dcslab.snu.ac.kr/projects/mpichgf/, 2004.