IEEE Copyright Notice

© IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Towards Easy-to-Use Checkpointing of MPI Applications within CLUSTERIX

Pawel Czarnul and Arkadiusz Urbaniak and Marcin Fraczak
Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, 11/12 Narutowicza, 80-952 Gdansk, Poland
{pczarnul,arkur}@eti.pg.gda.pl, marcin.f@wp.pl

Maciej Dyczkowski and Bartłomiej Balcerek
Wroclaw Centre for Networking and Supercomputing, Wroclaw University of Technology
{maciej.dyczkowski,bartlomiej.balcerek}@pwr.wroc.pl

Abstract

While there exist many kernel and user level libraries/systems which support checkpointing working processes and resuming their operations, it is still very difficult to provide an easy-to-use tool to assist checkpointing parallel applications. In this work, we aim at the development of an easy-to-use user-guided library to support checkpointing parallel MPI applications to be executed within the CLUSTERIX environment i.e. a collection of distributed HPC clusters. We propose a programmer-assisted approach with process state packing and unpacking at the code level for SPMD HPC applications. Although the library is in its early stage of development we present checkpoint/restart times and application execution (interrupted by checkpointing) times for the proposed approach compared to the same application linked with the ckpt user level library.

Keywords: Process Checkpointing, Checkpointing Parallel Applications, Parallel Software Environments

1. Introduction

There exist a number of solutions available for checkpointing ([4]) running processes, possibly done periodically, in order to suspend a process and possibly resume the operation afterwards, either on the same or another machine. The need for such an approach results from fault tolerance requirements, performance reasons i.e. effective process migration through checkpointing and resumption and/or requirements for equipment maintenance. In this work we focus on extending parallel MPI applications with a checkpointing module to be run on 64-bit clusters within the CLUSTERIX project ([1]). We assume the SPMD model of an application where the code iterates through a time loop where each process updates its subdomain, exchanges border data with its neighbors and proceeds with following iterations. This assumption simplifies the code development and makes it reasonable for, probably, the largest class of parallel time-consuming applications like FDM, MRTD and other codes ([20], [9]).

2. Related Work

In the literature a few main checkpointing techniques ([4]) are considered:

1. Compiler based checkpointing ([18]).
2. Kernel-level checkpointing ([11]). Provides automatic recovery of an application without any programming effort.
3. User-level checkpointing ([6], [10], [13]). Provides support for checkpointing through a run-time library. While it can be fully or to some extent transparent to the user, there are limitations with respect to the calls of the process this method can deal with as socket connections, messages, system calls etc.
4. User-guided checkpointing ([17]). The checkpoint contents and the places where checkpoint should be taken have to be defined by the programmer.

In this paper, we concentrate on the latter one following the user-defined checkpointing terminology defined in [17]. A similar approach has been proposed in the Legion manual ([19]). DAMPVM ([8], [7]) uses this technique for PVM applications. Representative HPC users of the Academic Computer Center in Gdansk (TASK) agreed to incorporate the user-guided approach into their codes and to provide
some additional programming effort gaining portability of the user-guided approach (kernel, MPI flavor independent) and possibly the smallest possible state size.

Examples of user-level checkpointing are Condor ([6]), Libckpt ([10]) and CoCheck ([13]). Other approaches include the Dynamite checkpointer 2.0 ([14]), esky ([2]), focus on LAM MPI extensions ([16], [15], LAM/mpi contains an interface which enables to use it with the kernel-level BLCR package for checkpointing parallel applications), migratable sockets ([5]), the EPCKPT library ([12]) includes support for checkpoint fork parallel applications. PORCH ([18]) is a source-to-source compiler for generating an equivalent program able to restart from a checkpoint.

3. Proposed User-assisted Checkpointing

```c
#include "mpickpt.h"

int nDoCheckpointing=0; // set to 1 after checkpointing
void *____mem; // request signal has been intercepted
long *____membytecount;
char *____processstatefilename; // name of the file
// storing a previously saved state of the process
int Initialize(int argc,char **argv)
{ FILE *file;
  // register a handler to a signal checkpointing process
  signal(SIGUSR1, CheckpointingRequestHandler);
  if ( (restarted copy)
    { // read checkpoint from file
      file=fopen(checkpoint.state,"rb");
      fread((void *)____membytecount,sizeof(long),1,file);
      // allocate memory
      ____mem=malloc(sizeof____mem);
      fread(____mem,1,____membytecount,file);
      fclose(file);
      return(EXIT_SUCCESS);
    }
  }

void CheckpointingRequestHandler(int state)
{ FILE *file; nDoCheckpointing=1;
  // IN THE TEST:
  // get the data from the user function and dump to file
  PackProcessState(____mem,____membytecount);
  // now write the data to a file
  file=fopen(checkpoint.state,"wb");
  fwrite((void *)____membytecount,sizeof(long),1,file);
  fwrite(____mem,1,____membytecount,file);
  fclose(file);
  MPI_Finalize(); exit(-1);
  // END IN THE TEST
}

int CheckAndExecuteCheckpointing()
{ if (nDoCheckpointing)
    { execute checkpointing
      synchronize all processes by sending MPI messages
      int nCode=PackProcessState(____mem,____membytecount)
    }
}
```

**Figure 1. Simplified mpickpt.c**

The checkpointing mechanism described in this paper is being designed to interact with the resource broker within CLUSTERIX. The CLUSTERIX architecture assumes that the existing resource broker implementation, GRMS (Grid Resource Management System) will be used, which was developed under the Gridlab Project ([3]). High level communication and control routines have already been defined there and are based on Java Web Services technology.

Following the serialization idea in Java, process state packing in DAMPVM ([8]), also proposed in other systems ([19], [17]), we have assumed that a process registers a handler function which intercepts signals SIGUSR1 which correspond to process state serialization requests. The compiled code of the MPI process is linked with a special library (Figure 1) which provides the standard application with several functions like the checkpointing initialization (Initialize() to be called before MPI_Init()), a function (CheckAndExecuteCheckpointing()) which checks whether a signal has been intercepted in CheckpointingRequestHandler(). For convenience, this library invokes two functions packing (PackProcessState()) and unpacking the process state (UnpackProcessState()) respectively which are to be supported by the programmer in the main code file of the application (Figure 2).

As shown in Figure 2, in the aforementioned assumed SPMD programming model, one can achieve the consistent state of the whole parallel application across all the nodes provided that all pending communication actions have been completed followed by packing individual process states (if a signal has been caught, this is done by function CheckAndExecuteCheckpointing() as shown in Figure 2). However, all MPI processes must be synchronized before similarly to the checkpointing approach of LAM/mpi as well. All processes proceed with saving their states to files.

```c
#include "mpickpt.h"

double *tab;
void PackProcessState(void *mem,long *membytecount)
{ *mem=malloc(sizeof(mem));
  *membytecount=malloc(sizeof(double));
}

int main(int argc,char **argv)
{ int gsize,mynumx,mynumy;
  MPI_Comm_size(MPI_COMM_WORLD,&gsize);
  MPI_Comm_rank(MPI_COMM_WORLD,&mynumx);
  // iterations through time
  for (;(tab++)<MAX_TIME_STEP;)
  
  #ifdef TEST FOR SINGLE_CKPT
  if ( (restarted) exit(-1); else send SIGUSR1 to myself to be checkpointed
  
  #endif
  
  perform computations on your part of the domain here
  CheckAndExecuteCheckpointing(); // could
  // be hidden in a wrapper to MPI_RECV/MPI_SEND
  MPI_Finalize();free(tab);
  return(EXIT_SUCCESS);
}
```

**Figure 2. Modified Application Code for Use with Checkpointing Library**
Resuming the application means restarting the MPI application with the same parameters as the original run extended with the state file name, also the number of processes started which has been stored in the descriptor of the application. As shown in Figure 1, the application uses library function `Initialize()` to reset the values of the variables using the user-supported deserialization function.

4. Experimental Results

Although the code is in its early stage of development, we aimed at the performance evaluation (for the measurement of checkpoint/restart procedure times, we used a single process application (calling MPI though) shown in Figure 2) and comparison of:

1. the proposed user-guided (serialized) checkpointing library developed by ourselves; the process sends a signal to itself to checkpoint itself; the process is resumed when given an additional command-line parameter,

2. the code linked with the `ckpt` library which provides the process with automatic checkpoint upon receiving a signal and followed by a restart of the process.

Checkpoint time is measured until the state file has been saved, executing same code in the two versions.

On the other hand, in this very case version 1 needs to restart its `main()` function executing initialization instructions in order to reach the main time step loop of our model SPMD application. The restart time is measured until the application completes. In the case of version 1 the code needs to reinitialize again. Since in this version the application terminates quickly (DEFINED in Figure 2) this favors version 2 given that the application packs practically all of its data in both cases, not giving an opportunity for version 1 to save on the checkpoint size.

The measured checkpoint/restart times are shown in Figures 3 and 4 showing similar results. However, in the cases when version 1 could save less data ([17], [8]), it would result in better times e.g. as seen by comparing “CKPT” for data size $d$ and “Serialization” for $d^2$ in Figure 3.

Moreover, to simulate a real world scenario, we measured the execution time of an application (in the two aforementioned versions) with a complete time stepping loop interrupted/resumed by/from several checkpoints through its execution. Achieved results are shown in Figure 5. One can see that the results are similar again, mainly due to the need for packing all of the application’s data. For large state sizes, the virtual memory issues show up. We used an Athlon XP 1800+ system with 256MBs of RAM.

5. Summary and Future Work

We presented a preliminary idea of user-guided checkpointing to be used in the CLUSTERIX project. The main goal is that the library should be independent from the op-

Figure 3. Comparison of checkpoint-restart time for different checkpoint methods and application data size. Checkpointing/restarting single application

Figure 4. Comparison of checkpoint and restart time for different checkpoint methods and application data size. Checkpointing/restarting multiple applications
Another version we are currently considering would use the ckpt library ([21]) to restart processes. Another MPI application with processes on same nodes can be started and the modified MPI communication functions of the previous process can allow the restarted MPI process to access the buffers (e.g. through shared memory) and instruct the new process to send/receive data through the new MPI.

Furthermore, we are looking at the checkpointing options available in LAM/MPi to be used for our project.

References

[6] Condor Team, Attention: Professor Miron Livny, Dept of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu, Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. The Condor Project, Condors Checkpoint Mechanism.

Figure 5. Execution time of the test application depending on number and method (serialization vs ckpt library) of checkpoints