On designing portable checkpointing tools for large-scale parallel applications

Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño, Ramón Doallo

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, Spain
{grodriguez, mariam, pglez, juan, doallo}@udc.es

Abstract- **Large-scale computational science and engineering parallel applications introduce the need for techniques that ensure that not all computation done is lost on machine failures. Application checkpointing has become a widely used approach to obtain such guarantees. This paper presents CPPC (Controller/Precompiler for Portable Checkpointing), an implementation that illustrates the use of compile time analysis, along with a runtime library, to obtain an efficient, scalable and portable checkpointing tool. It allows parallel processes to checkpoint independently, without runtime coordination or message-logging. Consistency is achieved at restart time by negotiating the restart point. CPPC was designed to work with parallel MPI programs, though it can be used with sequential ones, and easily extended to parallel programs written using different message-passing libraries, due to its highly modular design. Experimental results are shown using CPPC with different test applications.**

1. Introduction

Parallel computing evolution towards cluster and Grid infrastructures has created new fault tolerance needs. As parallel machines increase their number of processors, so does the failure rate of the global system. This is not a problem while the mean time to complete an application's execution remains well under the mean time to failure (MTTF) of the underlying hardware, but that is not always true on applications with long runs. For instance, let the MTTF of a single node on a Grid System be t_{node}. Assuming an application uses n nodes of the Grid, and using an exponentially distributed failure model, without fault tolerance, the probability of application termination would be:

$$f_n(t) = e^{-\frac{n}{t_{node}} \cdot t}$$

Where t should be replaced by the mean time to complete (MTTC) of the application. Suppose a Grid with an MTTF per node of 2 weeks, and that an application was executed over 8 nodes of the Grid with an MTTC of 2 days, then the resulting probability of completion would be 0.31. Therefore, users and programmers need a way to ensure that not all computation done is lost on machine failures.

Checkpointing has become a widely used technique to obtain such guarantees. It provides fault tolerance by periodically saving the computation state to stable storage, so that this state can be restored in case of execution failure. A number of solutions and techniques have been proposed [1], each having its own pros and cons. There are some significant drawbacks that are at least partially present in all existing solutions, each one tied to a remarkable property of general checkpointing techniques:

Granularity: Checkpointing can be studied from two different granularity levels, *data segment level* and *variable level*. On data segment level the entire application state is saved, recovering it when necessary. This approach has a general advantage: its independence of the considered application, since it is seen as a black box. But the more state it stores the less efficient the technique will be. Moreover, saving the application state entirely leads to lack of portability, as a number of non-portable structures will be saved along with application data (as application stack or heap).

Generally speaking, not all the state is needed when restarting an application, but there are certain parts that can be recovered based on a fundamental core of data. This leads to variable level checkpointing, which saves only restart-relevant state to stable storage. This approach is more efficient than data segment level techniques, and potentially generates portable files. The main draw-

back is the need of application analysis to identify the restart-relevant state, generally responsibility of the user, which results in a lack of transparency.

Scalability: Distributed checkpointing protocols are based on coordination or message-logging to ensure the consistency of the application state on recovery [2]. Both approaches reduce the scalability of checkpointing protocols. Even if processes log only received or sent messages, increasing the number of processes will multiply the number of flying messages, thus enlarging the computation needed per process to be protocol-compliant.

Portability: A state file is said to be *portable* [3] if it can be used to restart the computation on a different architecture (or OS) from that where the file was generated on. This means that state files should not contain hard machine-dependent state, which should be recovered at restart time using special protocols. Files saved using data segment level techniques will never be portable, as they store non-portable data, such as opaque MPI [4] state. Note that, as architecture (or OS) changes, so do MPI implementations, or even inter-process communication mechanisms.

Transparency: This property is measured in terms of user effort to implement a given solution. Traditionally, data segment level approaches are completely transparent to programmers, as they do not need much information about applications being treated. On the other hand, variable level views have to get some meta-data about the application in order to correctly operate, and they usually get it from the programmer. It seems to be a thin line separating both granularity and transparency levels of a given solution, but this line is an imaginary one, born from state-of-the-art trends. In fact, there are no limits for the information that the user can supply to a data segment level checkpointing tool, as there are not for the transparency that can be achieved by a variable level approach.

There are a significant number of decisions to be made upon facing the task of implementing a new checkpointing tool, all with advantages and disadvantages. This paper introduces CPPC (Controller/Precompiler for Portable Checkpointing), a checkpointing infrastructure that resolves major issues in implementing scalable, efficient and portable checkpointing mechanisms by using a variable level, non-coordinated, non-logging, portable checkpointing technique.

The structure of the paper is as follows. Section 2 gives an overview of CPPC's design and used techniques. Section 3 exposes CPPC's checkpoint operation mode. Section 4 presents the restart protocol used to recover the application state from a state file created during checkpoint operation. In Section 5 the operations needed to comply with both checkpoint operation and restart protocol constraints are shown. Fault tolerance is inserted into an example application using CPPC precompiler. Experimental results using CPPC's implementation are shown in Section 6. Related work is discussed in Section 7, and Section 8 concludes the paper.

2. CPPC Overview

Checkpointing tools based on data segment level techniques [5]-[8] usually have complex, non-portable implementations to deal with recovery of non-portable state. This makes restart impossible on different architectures. In fact, most variable level approaches [9, 11] store pieces of non-portable state. High performance computing trends towards heterogeneous clusters and Grid infrastructures make portability a desirable property. CPPC works at variable level storing only portable data, thus making possible to change the underlying architecture by using portable file formats, like HDF-5 [12]. Dumping format is fully configurable using writing plugins, thus allowing users to select (or develop) new ones.

Storing only portable data on state files has an important drawback, though. At restart time, not only variable data must be recovered, but also non-portable state created in the original execution (like MPI communicators, virtual topologies or derived data types). This introduces the need for some kind of restart protocol, capable of regenerating the original state that is not present on stored state files. CPPC uses code re-execution to achieve complete application state recovery. A piece of code is defined as *Required-Execution Code* (REC) if it must be re-executed at applica-

tion restart time to ensure correct state recovery.

When checkpointing parallel applications, special considerations regarding messages between processes have to be taken. There are two situations that require actions to be performed in order to achieve a correct restart:

- Process A sends message *m* to process B and takes its local checkpoint. Process B takes its local checkpoint before receiving *m*. If the application is restarted from that checkpoint, B will expect to receive *m*, but it will not be re-sent. Such messages are called in the literature *late*, *in-transit*, *in-flight* or *missing* messages.
- Process A takes its local checkpoint and sends message *m* to process B. B receives the message and takes its local checkpoint. If the application is restarted from that checkpoint, B will not expect to receive *m*, but it will still be re-sent. Such messages are called *early*, *orphan*, *inconsistent* or *ghost* messages.

A global checkpoint is said to be *transitless* if there are no in-transit messages when it is created [13]. It is called *consistent* if there are no ghost messages [14]. Traditional approaches use some kind of process coordination to ensure consistency, and message-logging techniques to solve issues generated by in-transit messages.

CPPC avoids overhead caused by coordination and message-logging by storing checkpoints at code points where it is guaranteed that there are no in-transit, nor ghost messages. Thus, generated checkpoints are transitless and consistent, both being preconditions for a checkpoint to be called *strongly consistent*. To identify such code points, which will be called *safe points*, CPPC relies on inter-process message flow analysis. Safe points can be easily found in well behaving SPMD codes. Such codes typically consist of a main loop which includes a communication phase. Messages between processes are typically sent and received in the same iteration, so just placing the checkpoint call before or after such a communication phase would suffice. Thus, strong consistency is achieved by using compile time coordination, rather than runtime one and message-logging, avoiding execution overhead.

Both working at variable level and placing checkpoint calls in safe points detected at compile time help achieve an efficient operation. Working at variable level reduces checkpoint file sizes, thus making state dump a light operation. Removing the need for message-logging or process coordination not only boosts scalability, but also efficiency. Besides, CPPC offers other checkpointing options, such as multithreaded dumping [15] and file compression. A scheme based on ZLib [16] has been developed, but other algorithms can be included. This does not only help saving disk space and network transfers (if needed), but also can improve performance when working with large datasets with high compression rates.

Using this approach, the user is responsible for detecting variables that must be recovered upon application restart (and thus must be stored as part of the checkpoint file) as well as safe points. RECs execution must be enforced using flow-control mechanisms that ensure that restart-relevant code is executed while disposable one is skipped. To ease this operation, a precompiler has been implemented so that users must only identify RECs and mark them using compiler directives. The precompiler is then responsible for inserting flow-control code. The SUIF framework [17] was used to implement the precompiler.

A general overview of CPPC's design can be seen in Figure 1. There are two different phases when using CPPC: compile time and runtime. At compile time, CPPC precompiler is used to transform a parallel application with user-inserted directives into a fault-tolerant parallel application. At runtime, the application will send petitions to the CPPC controller. As one of our objectives was to decouple CPPC controller from the specific host programming language, a single C++ implementation of the controller was written, while interfaces for different programming languages are provided. These interfaces are responsible for masking programming language features (such as static/dynamic memory management or variable typing) and forwarding petitions to the facade, which is responsible for managing the state that must be dumped when a checkpoint is reached. Thus, petitions regarding state to be dumped are fast, as they do not involve other controller layers. When the application asks for a checkpoint file to be created, the facade sends register information to the checkpointing layer, that formats these

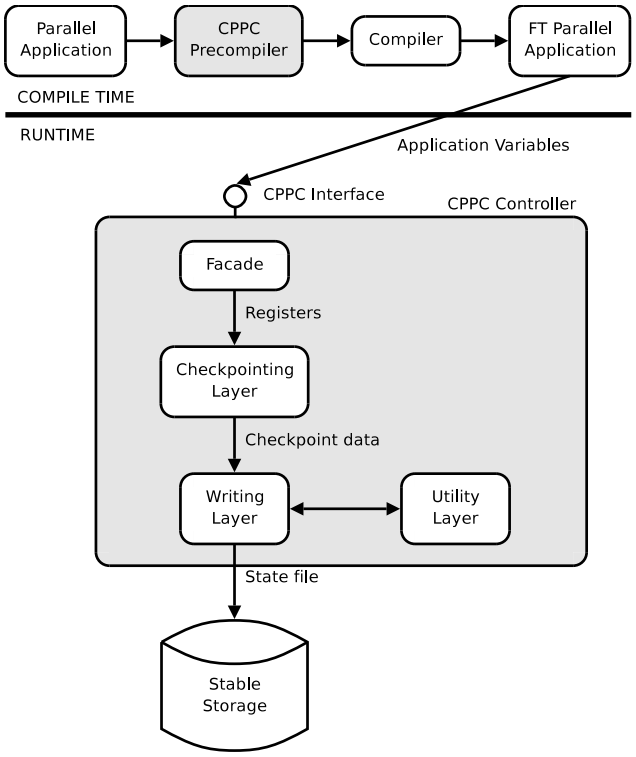

Figure 1: CPPC framework design

data so that the writing layer is able to process them. Every writing plugin must accept the same data format, so that they are fully interchangeable. When the writing layer receives a command to generate a checkpoint file, it may interact with the utility layer, where implementations of common algorithms are located (e.g. data compression or error detection codes generation). Finally, the writing plugin generates the checkpoint file containing the received data using its own format.

3. Checkpoint operation

CPPC has two operation modes: *checkpoint operation* and *restart operation*. The first one takes place when a normal execution is being run, while the second one emerges when the application must be restarted from a previously saved checkpoint file. This section covers the main five tasks that must be inserted into a normal execution flow to achieve fault tolerance using CPPC.

Initialization: First, a number of structures must be created in order for the controller to work correctly (such as a configuration manager, or the directories where CPPC will store checkpoint data), and contextual information must be gathered. Thus, an initialization function must be called by every process upon starting the parallel application.

Register: The main event in checkpoint operation is state dumping but, before this happens, CPPC must be aware of which parts of the process state must be stored, and which ones can be safely ignored. As mentioned before, CPPC stores only portable state, that is, user variables. Those to be stored are marked to achieve correct state dumping. In order to accomplish this, a *register* function, whose main goal is to provide such information to the CPPC controller, is needed. This function receives the data shown in the leaves of the checkpoint file hierarchy in Figure 2 and stores them as a tuple. CPPC will mark this memory region as relevant for dumping in subsequent checkpoints. The register function has other functionalities on restart operation covered in Section 4.

Unregister: It could happen that a memory region that was needed in previous execution points is not required anymore. To avoid dumping these "dead" variables CPPC provides an unregister function.

Checkpoint: At this point, CPPC has been correctly initialized and a checkpoint has been reached. All restart-needed memory regions are now marked for dumping, so that the operation can begin.

CPPC uses output plugins that are selected at runtime. Data to be stored are structured and passed to a writing plugin which will perform the effective data writing, using its own format. Besides, a writer must be able to read its generated files and restore register state and data. Users are allowed to implement new writing plugins that can be attached to the controller without recompilation. There is only one constraint in the format of the files being generated: the first byte must contain a unique identifier so that the CPPC controller can select the correct plugin for reading the file on restart operation. The input data hierarchy for every writing plugin is the same. It is shown in Figure 2. A checkpoint is composed of N sections, and a section contains M registers. Sections are used so that different registers may have the same identifier (variable name) in different sections. Currently, two writing plugins have been

implemented: a binary one, which dumps data in memory format thus allowing fast operation; and an HDF-5-based one, which makes it possible to move state files between machines in a heterogeneous cluster or Grid infrastructure.

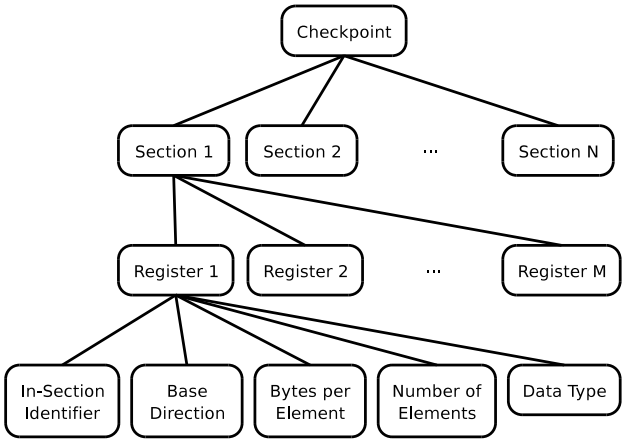

Figure 2: Checkpoint data hierarchy

As said in Section 2, state dumping can be done in a multithreaded way to improve performance. CPPC controller makes a copy of the memory that will be stored to avoid inconsistencies and then returns control to the parallel application, creating a new thread to handle checkpointing. Compression is also available, but only recommended to store data efficiently or if the compression rate is significantly high.

Shutdown: Finally, CPPC interface exposes a function to safely shutdown the system, making sure that checkpoints that are being stored concurrently are completed before freeing memory, and that working directories are left in a consistent state.

As can be seen, in order to reduce checkpoint overhead, thus enhancing efficiency, checkpoint operation was intended to be remarkably simple. As restart is not expected to be a common operation, complexity should be detached from checkpoint operation and moved to restart operation. The protocol used for recovery is exposed in the next section.

4. Restart operation

The restart operation comes across as a three-phase process. The first phase is a negotiation, where inter-process communications are performed so that an agreement about the restart point is achieved. During this phase, available checkpoint files are checked for errors, to ensure that no failures occurred during checkpointing in the original execution. No consistency issues are relevant as state files have been stored in safe points. The second phase consists of reading the selected file and loading its contents into memory. The third phase is the effective recovery of the application state. CPPC's approach is based on the existence, on a given parallel application, of five different REC types that build up the restart skeleton that must be re-executed. The first and second phases of the restart process are done in the initialization REC. The third one is distributed among the remaining ones. The following sections describe the conditions that must be fulfilled at compile time to remove consistency issues and ensure proper negotiation of the restart point, the different REC types and the flow control mechanisms used to interconnect them.

4.1. Guaranteeing global consistency

The set of files that parallel processes select as restart data must build up a *strongly consistent global state*. One such state can be achieved if all processes dump their data at the same point in the code, and at the same point in the execution (for example, if the checkpoint is into a loop all processes must dump their state on the same loop iteration). CPPC's approach takes the following steps to guarantee that the recovered global state is strongly consistent:

- All processes execute the same number of calls to the checkpointing function at the same relative points. This does not mean that checkpoints can not be into conditional structures, like *ifs*, but that if there is a checkpoint in the *then* part then there must be another one in the *else* part.
- Dumping frequency is not a function of time, but of the number of calls made to the checkpoint function. That is, a state file is generated every N calls, being N user-defined.

If both conditions are fulfilled, then all processes will dump their state at the same points of the execution. Note that this does not mean that processes are synchronized.

If every state file is uniquely identified by a number (being incremented on every call), then the same identifier on different processes refers to a state file created at the same relative execution point, thus allowing processes to coherently argue about valid restart positions.

4.2. REC types

The five types of REC are:

Initialization: First, the initialization function must be called so that CPPC controller can obtain information about the current execution and initialize needed structures. Upon restart, processes must search for state files created in the past. Then, the negotiation is performed and the selected file is read. Negotiation's pseudo-code is shown in Figure 3. This kind of REC will appear only once in every application. It must be located at the very beginning of the execution so that no state-dependent operations are performed.

```
A = Available files, ordered by uni-
 que file code

agreement ← false
While not agreement
 N ← Newest correct checkpoint in A
 If all processes propose N then
  agreement ← true
 Else
  O ← Older restart point proposed
  NO ← {x ∈ A / x is newer than O}
  A ← A--NO
 End If
End While

Delete files older than N
```

Figure 3: Pseudo-code of the negotiation about the restart point

Memory recovery: User variables stored as memory registers need to be recovered when restarting the application in order to rebuild its entire state. Memory will be recovered independently from other processes, as data were dumped in a distributed way. In order to avoid memory allocation problems, data can not be recovered at once when executing CPPC's initialization, but a more subtle way is employed: the *register* function recovers data to its proper location. It receives all the information needed to identify not only the original register, but also where the memory is allocated in the current execution (through the base address parameter). Pseudo-code in restart operation is detailed in Figure 4.

```
M = Register's base address
C = Contents of the register in the
 checkpoint file
staticMem = Memory is static

If staticMem then
 memcpy(M,C)
 delete(C)
Else
 M ← &C
End If

Add register using M as base address
```

Figure 4: Register function's pseudo-code in restart operation mode

Unregister calls: As seen in Figure 4, register state is recovered when restarting an application. If a register was removed during the original execution, then it must be removed when restarting the application. Thus, unregister function calls that were executed in the original run must be re-executed at restart time.

Non-portable state recovery: Besides user variables, non-portable state must be recovered (e.g. MPI communicators). Mandatory execution blocks are pieces of the original code that create this non-portable state. They must be re-executed upon restart in order to recover such state. This kind of blocks can also be used to recover state that is indeed portable, but that the programmer decides to recalculate, probably because it enhances execution performance (e.g. generating large datasets may be faster than reading them from disk).

Checkpoints: Checkpoint function calls must be re-executed so that CPPC can evaluate the execution state to check if restart is complete. If so, the application must leave restart operation and enter checkpoint operation mode. Note that restart may only finish at checkpoint calls, as it is there where the original execution's state was dumped. There are two premises that must be fulfilled in order to consider restart finished:

- All variables contained in the state file have been recovered.
- The checkpoint is the same where the state file was originally created.

If restart is finished, then the execution mode changes to checkpoint operation by deactivating conditional jumps (see Section 4.3).

4.3. Restart execution flow control

CPPC's restart operation consists mainly in an ordered execution of RECs. This means that the execution flow must be able to jump from the end of one REC to the beginning of the next one. To accomplish this task, the precompiler inserts two types of code blocks:

- **Conditional jumps at REC end:** These are jumps that are only carried out when recovery is taking place, not during checkpoint operation. As jumps can not be done from one procedure into another (this would eliminate the context change thus creating memory inconsistencies), they must be inserted into the original application code.
- **Jump labels at REC beginning:** These labels serve as jump destination for conditional jumps (not inserted before the CPPC initialization REC).

Together, these two constructs build up the flow-control restart mechanism. The precompiler also inserts jump labels before a call to a function containing RECs so that they are executed on restart operation, as well as conditional jumps after the call.

5. CPPC Directives

A set of six directives is provided to fulfill the requirements generated by the CPPC approach. They are not a description in a specific programming language, but a semantic one. Table 1 summarizes directive purposes for each operation mode.

- `cppc init`: Used for marking the initialization point, where memory structures needed by the CPPC controller are initialized. It is also the beginning of the restart execution flow, where an agreement is reached and the selected checkpoint file is read into memory.
- `cppc shutdown`: For marking the CPPC finalization point, freeing used memory and ensuring that checkpoints being currently stored are correctly finished.
- `cppc register ( var1[size1], ... )`: Accomplishes a double functionality: on checkpoint operation updates the register state, which will be dumped at checkpoints. On restart operation it is the point where memory recovery is done.
- `cppc unregister ( var1, var2, ... )`: For unregistering variables that are not needed anymore. Execution on restart is needed to achieve correct register state.
- `cppc execute/end execute`: For marking mandatory execution blocks which achieve non-portable state recovery, such as calls to MPI functions that create communicators (e.g. `MPI_Comm_split()`).
- `cppc checkpoint`: Points where the state is dumped, or where it is checked whether restart has finished or not, depending on the operation mode.

Directive	Checkpoint op.	Restart op.
Init	Initialization	Initialization Negotiation Read checkpoint file
Shutdown	Achieve consistency Free used memory	Not used
Register	Create new register	Recover data Create new register
Unregister	Delete existent register	
Execute	Not used	Execute marked code
Checkpoint	Dump state file	Check restart state

Table 1: CPPC's interface directives summary

5.1. A CPPC application example

C and Fortran 77 versions of CPPC interface have been implemented to help reuse the same controller implementation (see Figure 1). For illustrative purposes, Figure 5 details the C code needed to transform an application (matrix diagonalization) into a fault tolerant version using CPPC. On checkpoint operation the MPI environment is initialized, then the CPPC controller; matrix data are read; inter-process communication is

started; matrix data are distributed and then registered by every process; next, the core computation of the application begins (with checkpoints taken every N iterations, being N user-defined); unneeded data are unregistered; then another checkpoint is taken and, finally, results are written and CPPC shutdowns. Upon restart, the execution flows as follows:

```
int main( int argsc, char **argsv ){

 double * matrix;
 int matrix_size, i, niters;
 char * matrixf;

 MPI_Init( &argsc, &argsv );
 #pragma cppc init

 /* Command-line parameters */
 matrix_size = atoi( args[1] );
 matrixf = args[2];

 /* Matrix data input */
 matrix = mread( matrix_size,
  matrixf );

 #pragma cppc execute
 /* Inter-process communication
  initialization */
 MPI_Comm_split( ...  );
 ...
 #pragma cppc end execute

 /* Matrix data distribution */
 ...

 #pragma cppc register ( i, niters,
  matrix[ matrix_size ] )
 for( i = 0; i < niters; i++ ) {
  #pragma cppc checkpoint
  /* Matrix diagonalization */
  ...
 }

 #pragma cppc unregister (i,niters)
 #pragma cppc checkpoint
 /* Diagonalized matrix output */
 ...

 #pragma cppc shutdown
 MPI_Finalize();
}
```

Figure 5: Fault-tolerant matrix diagonalization through CPPC directives

- Execution of the MPI initialization.

- Execution of the CPPC controller initialization. Processes agree about the restart file and read the checkpoint file. The directive translation is:

```
CPPC_Init( &argsc, &argsv );
/* Conditional jump to next REC */
if( CPPC_Jump_next() ) {
 next_jump = ( next_jump + 1 ) %
  labels_count;
 goto * jump_labels[ jump_index ];
}
```

  The precompiler automatically inserts the `jump_labels` array (to store jump destination labels), the `jump_index` integer (to index the `jump_labels` array), and the `labels_count` integer (which stores the size of the `jump_labels` array so that `jump_index` can be incremented coherently).

  `CPPC_Jump_next()` returns true if CPPC is on restart operation mode, meaning that conditional jumps must be taken. These jumps are always translated the same way, so this code will not be repeated below.

- Mandatory execution of the inter-process communication initialization block:

```
CPPC_MANDATORY_BLOCK_1:
/* Inter-process communication
   initialization */
...
/* Conditional jump to next REC */
...
```

- Execution of the variable data recovery block (register directive). Note that loop index and limits are registered so that they are preserved through executions:

```
CPPC_REGISTER_BLOCK_1:
CPPC_Register( 0, &i, 1, CPPC_INT,
 ``i'', CPPC_STATIC);
CPPC_Register( 0, &niters, 1,
 CPPC_INT, ``niters'',
 CPPC_STATIC );
CPPC_Register( 0, &matrix_size, 1,
 CPPC_INT, ``matrix_size'',
 CPPC_STATIC );
matrix = CPPC_Register( 0, matrix,
 matrix_size, CPPC_DOUBLE,
 ``matrix'', CPPC_DYNAMIC );
/* Conditional jump to next REC */
...
```

- Execution of the first checkpoint call:

```
CPPC_CHECKPOINT_1:
CPPC_Do_checkpoint( 0 );
/* Conditional jump to next REC */
...
```

  The checkpoint function tests the two conditions to enter checkpoint operation seen in Section 4.2. The first one (all state is recovered) will always be true in this example, as all register functions have been executed. If the selected state file was generated at this point in the original execution then restart is over, and `CPPC_Jump_next()` will return 0 from now on. If it was generated at the second checkpoint then the conditional jump is taken and restart continues. Every checkpoint call receives a parameter, automatically generated by the precompiler, that serves as unique identifier so that CPPC is able to distinguish between different checkpoint calls.

- Execution of the block that contains only variable unregistration:

```
CPPC_UNREGISTER_BLOCK_1:
CPPC_Unregister( &i );
CPPC_Unregister( &niters );
/* Conditional jump to next REC */
...
```

  Unregister calls must be executed at restart time in order to correctly restore registers state. If not done, then registers that had been already removed in the original execution would remain active when restarting.

- Execution of the second checkpoint call. Translation is similar to the first one, except that the unique identifier passed to the function `CPPC_Do_checkpoint()` will be 1, not 0. At this point the execution will enter checkpoint operation, as the original state is now completely restored.

6. Experimental results

The NAS Parallel Benchmarks (NPB) [18] were used to test CPPC. Five of these applications have been selected: BT (Block Tridiagonal), CG (Conjugate Gradient), LU (Lower-Upper Symmetric Gauss-Seidel LU), MG (MultiGrid) and IS (Integer Sort), as they are a representative set because of their different state file sizes. Also, IS was interesting because it is the only C application in the NPB. Executions took place on a cluster of four biprocessor Intel Xeon nodes, 1.8 Ghz, 1 GB of RAM, connected through a SCI network. Size of the state files being generated, the overhead for checkpoint operation and the restart time were measured.

6.1. State file sizes

When using a non-coordinated, non-logging checkpointing technique, it is clear that the overhead introduced will only depend on checkpoint sizes. Thus, the first parameter to be measured is how the variable level approach affects checkpoint file sizes. Results for the test applications are shown in Figure 6, using the binary writer, binary writer with ZLib compression and HDF-5 writer. Variable level checkpointing achieves very important size reductions on some applications when compared to segment level, like in BT where it reaches 80%.

Note that the HDF-5 writer generates files approximately the same size as those generated by the binary writer. HDF-5 library is configured to dump data in memory format. Conversions, if needed, will be done at restart time. Thus, CPPC-generated HDF-5 files are much like the binary ones, except that all data are tagged (to make conversions possible when restarting) and structured in a different way. The reasons for delaying conversions until restart are that they may not even be necessary, and that this improves performance of checkpoint operation.

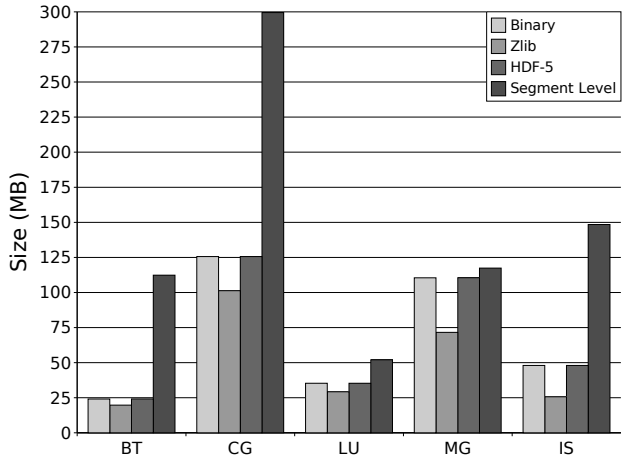

Figure 6: Generated per-node file sizes

6.2. Checkpoint overhead

As said before, the performance obtained by CPPC is tightly tied to the size of the files being generated. The other factor that plays a key role in dumping time is the algorithm being used. Figure 7 details dumping times.

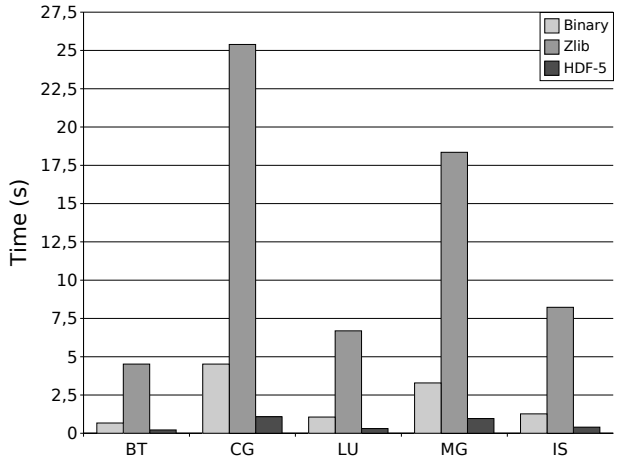

Figure 7: File generation overhead

Note that the HDF-5 writer shows better performance than the binary one. There are two reasons for this. First, no data conversion is being made on checkpoint operation. And second, the HDF-5 writer does not implement any error detection code scheme for complete files, while the binary writer includes CRC-32 (Cyclic Redundancy Check, 32 bits).

It is clear that using compression greatly increases dumping time. Thus, its use is only recommended if effective gains are going to be obtained, for example, if there are problems with disc quotas or very high compression rates.

To avoid the overhead introduced by file generation, multithreaded state dumping has been implemented. Table 2 details the original execution times (without checkpointing) and the overhead introduced by the generation of a single checkpoint file using the binary writer. Good results are obtained with those applications with higher runtimes. MG and IS have such a short execution that creating a state file generates a high overhead (although this overhead is in the range of 1-2 seconds). Note that checkpoint overheads include, besides state dumping, the remaining CPPC operations: initialization, variable registration/unregistration and finalization.

	Original runtime	**Checkpt. overhead**	**Overhead percentage**
BT	638.37	2.70	0.42%
CG	617.28	7.27	1.18%
LU	467.08	4.7	1.01%
MG	21.47	2.05	9.55%
IS	5.72	1.50	26.22%

Table 2: Multithreaded checkpoint overheads in seconds

A lot of useful information can be extracted from these applications, even when their execution times are around minutes (making the checkpointing period not acceptable). However, these results can be extrapolated to infer the total checkpointing overhead in any given application, known its execution time and generated file sizes. Thus, if an application runs for 30 days, using CPPC with a checkpointing period of 1 day and assuming checkpoint files of 2 GB per node would only increase estimated execution time by 2 hours, which is rather an acceptable time (it represents an overhead of 0.27%).

6.3. Restart overhead

Execution overhead has been measured assuming that there are no failures during the execution. If this happened, restart time would play a fundamental role in total execution time. Restart times for the previous applications have been measured and split to its 3 fundamental parts: negotiation, file reading and effective recovery. Results can be seen in Figure 8. Negotiation time and file reading time greatly depend on file sizes, as there is the need to check them for errors using the previously

generated CRC-32 code. Also, both depend on the plugin that generated the file: using the ZLib writer negotiation time is reduced (as the CRC-32 is performed over smaller files), but reading time is increased (as it is necessary to decompress the file); using the HDF-5 writer negotiation time almost vanishes (as no integrity tests are performed) and reading time could increase (if data transformations are needed). Recovery time only depends on the amount of state saved and the amount of state recovered using code re-execution and represents a minor percentage of the total restart time. As can be seen, restart times are rather low, making restart overhead negligible except when dealing with applications with extremely short runs, such as MG or IS.

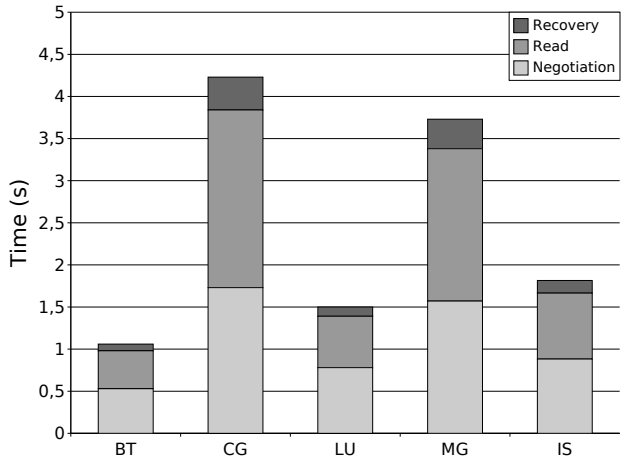

Figure 8: Restart times using CPPC's binary writer

7. Related work

There are currently available several solutions that deal with checkpointing. The main difference between all other implementations and CPPC is portability, whether code portability or checkpoint file portability.

PC/MPI [11] focuses on code portability. It is implemented over a sequential checkpointer like CKPT [19]. Hence, it works at segment level, generating non-portable files. For global consistency uses the same safe point approach as CPPC.

MPICH-GF [6] and MPICH-V2 [7] are implemented as MPICH [20] drivers, thus enforcing all machines to run a given MPI implementation. Both work at data segment level, thus generating non-portable files. MPICH-GF uses process coordination, while MPICH-V2 logs sent messages. Both are non-scalable approaches to achieve global consistency.

CLIP [5] is a Libckpt-based [21] implementation for Intel Paragon architectures. File portability is a non-sense in this context. It also uses process coordination to deal with global consistency, flushing message buffers and storing them along with the state file.

Dome [22] extensions achieve to perform portable checkpointing [23], as they use the Dome environment for data managing, which is natively portable. This solution can only be used when working with Dome programs. Processes may checkpoint independently from others and in any given point in the program without using global consistency techniques, as communications in a Dome program are implicit. This discards the possibility of in-transit or ghost messages while dumping data.

CoCheck [8] was developed for Condor system [24]. It works at segment level, thus discarding file portability. It uses process coordination to achieve global consistency, thus being a non-scalable approach.

C^3 [9, 10] is a checkpoint compiler that works at variable level, implemented over the MPI library. Thus, its code is independent of the MPI implementation. It uses a newly developed protocol based on the Chandy-Lamport protocol [13] to achieve global consistency. Scalability of this protocol is under study. Portability is not one of its goals and, although it does not store any internal MPI state, enforces data to be recovered on the same virtual location than on the original execution to achieve pointer consistency, thus making impossible architectural changes and file portability.

8. Concluding remarks and future work

In this paper a new checkpointing infrastructure, CPPC, has been presented and evaluated. CPPC resolves major issues in implementing scalable, efficient and portable checkpointing by using variable level, non-coordinated, non-logging, portable approaches.

From a global view, CPPC's most remarkable property is the portability of the files being generated, as well as of its source code. This is an interesting characteristic due to inherent hetero-

geneity of Grid infrastructures. Dumping format is fully configurable using writing plugins, being possible to include the developments done by the Grid and Recovery Group at the Global Grid Forum (Grid-CPR) [25]. The introduction of the precompiler tool greatly reduces the programmer effort, as it removes the need to write tedious, mechanical code.

A related work section has been included. Unfortunately, a comparative performance analysis between CPPC and other solutions has not been possible due to lack of implementations available on the web.

The drawback of the safe point approach is that inter-process message flow must be analyzed for every application. However, this is not a hard task and, more importantly, it can be automatized. We are currently working on techniques for automatically or semi-automatically extracting information about safe points, relevant variable analysis and code blocks that must be re-executed upon restart, in order to obtain a more transparent approach to checkpointing.

Acknowledgments

This research was supported by the Ministry of Education of Spain (Project TIN2004-07797-C02) and by Xunta de Galicia (Project PGIDIT04TIC105004PR).

References

[1] E. N. Elnozahy, L. Alvisi, Yi-Min W., and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[2] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *Proceedings of the 2003 IEEE International Conference on Cluster Computing (CLUSTER'03)*, Hong Kong, Dec. 2003, pp. 242–250, IEEE Press.

[3] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogenous architectures," in *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, Washington - Brussels - Tokyo, June 1997, pp. 58–67, IEEE Press.

[4] Message Passing Interface Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, Department of Computer Science, University of Tennessee, Apr. 1994.

[5] Y. Chen, J. S. Plank, and K. Li, "CLIP: A checkpointing tool for message-passing parallel programs," in *Proceedings of the 1997 ACM/IEEE Conference on SuperComputing (SC'97)*, San Jose, CA, Nov. 1997, pp. 1–11.

[6] Namyoon W., Hyungsoo J., Heon Y. Y., Taesoon P., and Hyungwoo P., "MPICH-GF: Transparent checkpointing and rollback-recovery for Grid-enables processes," *IEICE Transactions on Information and Systems*, vol. E87-D, no. 7, pp. 1820–1828, 2004.

[7] A. Bouteiller, F. Capello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*, Phoenix, AZ, 2003.

[8] G. Stelnner, "Cocheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, HI, 1996, pp. 526–531, IEEE Press.

[9] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," in *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, San Diego, CA, June 2003, pp. 84–94, ACM.

[10] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant MPI," in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*, San Francisco, CA, June 2003, pp. 234–243, ACM.

[11] Sunil A., Jungwhan K., and Sagyong H., "PC/MPI: Design and implementation of a portable MPI checkpointer," in *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, Venice, Italy, Sept. 2003, vol. 2840 of *Lecture Notes in Computer Science*, pp. 302–308, Springer–Verlag.

[12] National Center for Supercomputing Applications, "HDF-5: File Format Specification," http://hdf.ncsa.uiuc.edu/HDF5/doc/.

[13] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[14] J-M Hélary, R. H. B. Netzer, and M. Raynal, "Consistency issues in distributed checkpoints," *IEEE Transactions on Software Engineering*, vol. 25, no. 2, pp. 274–281, March/Apr. 1999.

[15] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Dis-*

tributed Systems, vol. 5, no. 8, pp. 874–879, Aug. 1994.

[16] J. Gailly and M. Adler, "ZLib Home Page," http://www.gzip.org/zlib/.

[17] M. W. Hall, J.A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler system," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, Dec. 1996.

[18] National Aeronautics and Space Administration, "The NAS Parallel Benchmarks," http://www.nas.nasa.gov/Software/NPB/.

[19] V. C. Zandy, "CKPT Library Home Page," http://www.cs.wisc.edu/~zandy/ckpt.

[20] W. Gropp and E. Lusk, *User's guide for mpich, a portable implementation of MPI*, Argonne National Laboratory, University of Chicago, July 1996.

[21] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, New Orleans, LA, Jan. 1995, pp. 213–223.

[22] A. Beguelin, E. Seligman, and M. Starkey, "Dome: Distributed Object Migration Environment," Tech. Rep. CS-94-153, Carnegie-Melon University, 1994.

[23] A. Beguelin, E. Seligman, and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, 1997.

[24] M. Litzkow, M. Livny, and M. W. Mutka, "Condor – A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, San Jose, CA, June 1988, pp. 104–111, IEEE Press.

[25] Global Grid Forum, "Global Grid Forum Grid Checkpoint Recovery Working Group," http://gridcpr.psc.edu/GGF/.