Portable Checkpointing of MPI Applications*

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

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

Abstract. CPPC (Controller/Precompiler for Portable Checkpointing) is a checkpoint-based fault tolerance tool focused on the execution of parallel applications over heterogeneous clusters or Grids. In addition to using portable code, protocols and data storage formats, compile-time analysis of the application to be checkpointed is needed in order to avoid restart inconsistencies, without falling back to traditional approaches like process coordination or message-logging, which typically introduce unscalable overheads. These solutions are implemented into the CPPC library, used by the CPPC source-to-source preprocessor to automatically transform a parallel application into a fault tolerant counterpart. Experimental results to assess the checkpointing tool are also provided.

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, where 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 [7], each having its own pros and cons.

The most remarkable property of general checkpointing techniques is *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.

* This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN-2004-07797-C02) and by the Galician Government (Project PGIDIT04TIC105004PR).

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).

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 usage. Section 3 explains in more detail CPPC's behavior and implemented techniques. Section 4 details tasks that can be automatically performed by the CPPC precompiler. In Section 5 an example of how to use CPPC with a simple application is shown. Experimental results using CPPC are presented in Section 6. Related work is discussed in Section 7, and Section 8 concludes the paper.

2 CPPC Overview

CPPC appears to the user as a compiler tool and a runtime library which help achieve the goal of inserting fault tolerance into a parallel application. The user inserts compiler directives into the original application, and then runs the precompiler that performs a source-to-source transformation. The result is a fault tolerant code which makes calls to the CPPC library. Available directives are:

- **cppc init**: Used for marking the initialization point, where memory structures needed by the CPPC tool are initialized.
- **cppc shutdown**: For marking the CPPC ending point. It ensures that checkpoints being stored concurrently with the execution of the parallel application are left in a consistent state before exiting.
- **cppc register(var1[size1], ...)**: Used to mark variables that will be relevant upon restart, and hence must be stored in the checkpoint file.
- **cppc unregister(var1, var2, ...)**: For unregistering variables that are no longer needed.
- **cppc execute/end execute**: For marking mandatory execution blocks that achieve recovery of non-portable state, such as MPI communicators. Note that if this non-portable state were saved to disk it would prevent re-execution on different architectures or operative systems.
- **cppc checkpoint**: Points where the state is dumped. Here, registered variables will be saved to disk.

The insertion of the first four directives is automatically done by the precompiler, as will be explained in Section 4.

The CPPC precompiler is built on the Cetus [13] compiler infrastructure. It is written in Java, which helps achieve the portability objective in CPPC. Its intermediate representation language (IRL) allows for transformations to be performed using the same code for different source programming languages.

A general overview of CPPC's design can be seen in Figure 1. CPPC library calls are implemented through delegation into a controller. As one of the design goals was to decouple the 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 (currently supported languages are C and Fortran 77). These interfaces are responsible for masking programming language features (such as static/dynamic memory management or variable typing) and forwarding petitions to the facade, 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 data so that the writing layer is able to process it. There are different implementations for the writing layer. It determines the state file format and properties. The writing layer may interact with the utility layer, where implementations of common algorithms for checkpoint dumping are placed (such as data compression or generation of error detection codes). Finally, the writing plugin generates the state file containing the checkpoint data using its own format.

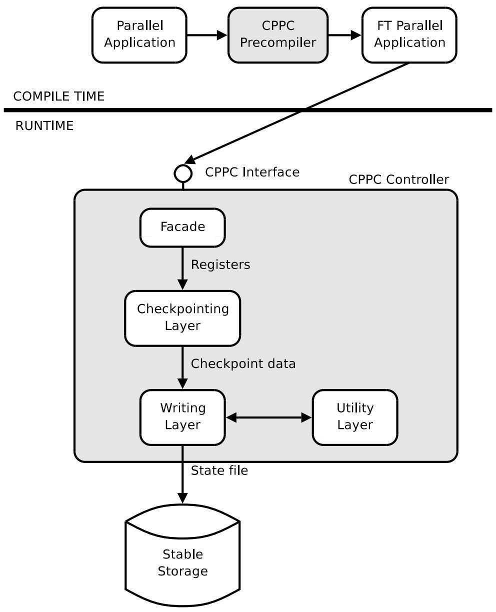

Fig. 1. CPPC framework design

3 Implementation details

CPPC has two operation modes: *checkpoint operation* and *restart operation*. The first one takes place when a normal execution is being run. It basically consists on registering relevant variables and dumping checkpoint files at marked locations in the code.

The second one emerges when the application must be restarted from a previously saved checkpoint file. Both portable and non-portable state have to be recovered. Also, when working with parallel applications it must be ensured that all processes are restarted in a consistent global state: no coordination problems may arise at restart time.

The following sections address the techniques used by the CPPC tool for dealing with the aforementioned issues.

3.1 Checkpoint file dumping

CPPC is a variable level checkpointer. The `cppc register` directive is used for marking those regions of portable memory (user variables) which have to be saved into checkpoint files. When the application reaches a checkpoint, data to be stored are structured and passed to the writing layer. The effective data writing will be performed by the selected writing plugin implementation. This enables the restart on different architectures, as long as a portable dumping format is used for code variables. A writing plugin must also be able to read its generated files and restore registers state and data. Users are allowed to implement new writing plugins that can be attached to the library without recompiling it.

Currently, two writing plugins have been implemented: a binary one, which dumps data in memory format thus allowing fast operation; and an HDF-5 [17] based one, which makes it possible to move state files between machines in a heterogeneous cluster or Grid infrastructure.

Besides, CPPC offers other checkpointing options, such as multithreaded dumping [14] and file compression. A scheme based on the ZLib library [8] has been developed, but other algorithms can be included (again without recompiling the library). This does not only help save disk space and network transfers (if needed), but can also improve performance when working with large datasets with high compression rates.

3.2 Non-portable state recovery

Storing only portable data on state files has an important drawback: at restart time, not only code variables must be recovered, but also non-portable state created in the original execution (such as 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 application restart time to ensure correct state recovery.

Through code re-execution, non-portable state is recovered by the same means used in the original execution. Therefore, a CPPC application is just as portable as the original one: variables are saved in a portable manner, non-portable state is recreated using the original code.

3.3 Global consistency

When checkpointing parallel applications, special considerations regarding messages between processes have to be taken to ensure that the coordination implicitly caused by the communication flow between processes is not lost when restarting the application. Therefore, global consistency of application processes must be guaranteed. 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, which 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 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 [4]. It is called *consistent* if there are no ghost messages [12]. 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 the overhead caused by coordination and message-logging by using spatial coordination, rather than temporal coordination, that is, checkpoints are taken at the same relative code points by all processes (assuming SPMD codes). This ensures that all processes can select the same relative code point for restarting the parallel application, instead of restarting from different points which would render the global state inconsistent. To avoid problems caused by messages between processes, checkpoint directives must be inserted at points where it is guaranteed that there are no in-transit, nor ghost messages. These points will be called safe points. Thus, generated checkpoints are transitless and consistent, both being conditions for a checkpoint to be called *strongly consistent* [12].

To automatically identify safe points, a static analysis of inter-process message flow would be needed. However, locating safe points is not a difficult task for the user (specially when dealing with SPMD applications), who is currently responsible for placing checkpoint directives at adequate points in the code. A reliable automatization of this task is not straightforward, since a static modelling of communications between parallel processes would be necessary. The feasibility of such an automatization is under study.

3.4 Restart protocol

The restart operation consists of three phases: negotiation, state file reading and application state recovery.

In the first one 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. Once a file is selected, the second phase consists of reading its contents into memory.

Finally, in the third phase, the application state must be effectively recovered. The technique used is to re-execute RECs in order: `init`, `register` and `checkpoint` directives, and code enclosed in between `execute/end execute` directives. CPPC has to be able to control the execution flow so that it jumps from the end of one REC to the beginning of the next one. To accomplish this task, the precompiler automatically inserts two types of code blocks:

- Conditional jumps at REC end: These are jumps that are only carried out when restart 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 inconsistent memory state), 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 init` REC).

Together, both constructs build up the flow control restart mechanism. The precompiler also inserts jump labels before calls to procedures containing RECs so that they are executed at restart, as well as conditional jumps after the call.

4 Automatic directive insertion

In addition to transforming user directives into fault tolerant code, the CPPC precompiler includes some automatized analysis to relieve the user from tedious tasks such as relevant variable analysis.

Currently, the `init`, `shutdown`, `register` and `unregister` directives are automatically inserted by the precompiler. The remaining ones, `checkpoint` and `execute/end execute`, are easily introduced by the user, as it is only required a basic idea of how the application code is organized. The `register` and `unregister` directives, though, require a mechanical data-flow analysis, which can be a tedious and error-prone task to be performed by the user.

The automatization of the `init` and `shutdown` directives insertion is straightforward: after and before calls to the parallel code initialization/shutdown functions (`MPI_Init/Finalize`), respectively.

In order to automatize `register/unregister` directives, the precompiler performs a live variable analysis [15] of application blocks which are subject to re-execution when the application has to be restarted. Such code blocks are the ones enclosed by `execute/end execute` directives, and the code following a `checkpoint` directive up to the end of the application (assuming that the execution will enter checkpoint operation mode after reaching that checkpoint). This is a somehow complementary approach to techniques used in sequential data segment level checkpointers to reduce the amount of memory stored, as the one proposed in [18].

When the relevant blocks of code have been identified, the precompiler has to generate the set USE of *used* variables and the set DEF of *defined* variables in such blocks. After generating these sets, the precompiler finds the set of live variables at the beginning of each of these code blocks, using the following expression:

$$LVin(B) = (LVout(B) - DEF(B)) \cup USE(B)$$

where $LVin(B)$ is the set of live variables when entering block B and $LVout(B)$ is the set of live variables when exiting block B. This analysis has to be performed backwards, since $LVout(B_{end}) = \emptyset$, being B_{end} the last block of code.

For each block B_n, whether it is an `execute/end execute` block or the code after a `checkpoint` directive, the precompiler inserts a register directive for the variables contained in the set $LVin(B_n)$. Variables registered or defined at previous blocks are not registered again. Also, before each checkpoint block B_c the precompiler inserts an unregister directive containing the variables in the set $LVin(B_{c-1}) - LVin(B_c)$, where B_{c-1} denotes the previous checkpoint block in the application code.

The precompiler is able to deal with pointer and array variables. However, it does not statically detect aliasing and it does not perform bounds checks. Aliasing is detected at runtime by the CPPC library, and the user is properly warned. Pointer storage techniques based on machine-independent offsets [22] are being considered to solve statically aliasing problems. Not performing bounds checks means that arrays and pointers are treated in a conservative and reliable way: they are entirely stored if they are used at any point in the re-executed code, and are never considered defined.

Note that `init`, `shutdown`, `register` and `unregister` directives can also be manually inserted by the user (e.g., to checkpoint only critical sections of code, or to optimize the size of state files when non live array sections are being stored due to the lack of bounds checks).

5 Example

For illustrative purposes, an example of how to insert fault tolerance through the use of CPPC directives into a code which plays Conway's Game of Life [9] is presented. Figure 2(a) shows the original, annotated code. There are basically three different sections in this code. The first one initializes local variables in each parallel process. The second one creates necessary MPI structures, such as communicators and data types for modelling the communication between processes. The third part is the computational loop, which calculates the board for the next "generation" in the game.

The first thing that needs to be annotated is the block which creates MPI structures and types. It must be re-executed, since MPI state is non-portable.

This way, CPPC ensures that all needed MPI structures are recovered in a portable manner: executing the same function calls that were used upon the original run. Also, a checkpoint is placed at the beginning of the main loop, which is a clear safe point.

Once the user has annotated the code, the CPPC precompiler begins to perform analysis and code transformations. As an intermediate step, the precompiler performs the automatic insertion of `init`, `shutdown`, and `register` directives, previously performing a data flow analysis of the application. No `unregister` directives are inserted because there is only one checkpoint directive in this example.

After inserting all directives, the precompiler is now ready to begin translating them. The resulting fault tolerant code can be seen in Figure 2(b). The precompiler inserts new variables which are used to control the execution flow on application restart, ensuring that only those relevant blocks of code are visited. Conditional jumps and jump destinations are placed enclosing each REC, in order to control the execution flow when restarting (see Section 3.4).

In terms of execution flow, when restarting the application, variables are initialized and both the MPI and CPPC initialization functions are called. Then, the conditional jump takes place, transferring execution flow to the `execute` REC. After recovering all MPI state, the execution flow moves to the `register` REC, recovering values for live variables. Finally, the execution flow reaches the checkpoint placed in the original code. CPPC determines that all state has been restored, and the execution is resumed normally.

6 Experimental results

For an adequate testing of the CPPC tool, a twofold approach was selected. On the one hand, the NAS Parallel Benchmarks (NPB) [16] were used. Five of these applications were 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 (the other ones are Fortran benchmarks). On the other hand, a civil engineering parallel application called Hipsia [11] which calculates stresses caused by a crack growth in a finite body was also added. Tests were performed on a cluster of four biprocessor Intel Xeon 1.8 Ghz nodes, 1 GB of RAM, connected through a SCI network. Size of generated state files, time overhead for checkpoint operation and restart time were measured.

6.1 State file sizes

When using a non-coordinated, non-logging checkpointing technique, it is clear that the incurred overhead 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 3, using the binary

```
int main( int argsc, char **argsv ){
  /* Variable definitions here */
  MPI_Init( &argsc, &argsv );

  MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
  MPI_Comm_size( MPI_COMM_WORLD, &worldsize );

  /* Check and obtain parameters */
  ...
  /* Initialize game boards */
  if( mytid == 0 ) {
    global_board = (char *)malloc( ... );
  }
  local_board = (char *)malloc(
    local_size * sizeof(char) );
  #pragma cppc execute
  /* Initialize communications and types */
  MPI_Cart_create( ..., &gridComm );
  MPI_Comm_rank( gridComm, &mytid_grid );
  MPI_Cart_coords( gridComm, ..., my_coords );
  /* Initialize here data types
   for comms such as boardRow */
  /* Find TIDs for adjacent processes
   in the cartesian grid */
  MPI_Cart_shift( gridComm, ..., &tid_up );
  ...
  #pragma cppc end execute

  while( 1 ) {
    #pragma cppc checkpoint
    /* Send local board ends to neighbors */
    MPI_Send( local_board, 1, gameRow, tid_up,
      0, gridComm );
    ...
    /* Receive from adjacent processes */
    MPI_Recv( local_board, 1, gameRow, tid_up,
    2, gridComm );
    ...
    /* Compute next local iteration */
    ...
    /* Gather all boards in a global
     one for displaying it */
    MPI_Gatherv( local_board, ..., ...,
      global_board, ..., 0, gridComm );
    if( mytid_grid == 0 ) {
      showBoard( globalBoard );
      /* Code prompting user for next
       iteration */
      ...
      option = getchar();
    }
    /* Broadcast user option */
    MPI_Bcast( &option, 1, MPI_CHAR, 0,
      commGrid );
    if( option == 'q' ) {
      MPI_Finalize();
      exit( 0 );
    }
  }
}
```

(a) CPPC-annotated Conway's game of life

```
int main( int argsc, char **argsv ){
  /* Variable definitions here */
  void * cppc_jump_points[ 3 ] = {
    &&CPPC_EXECUTE_BLOCK_0,
    &&CPPC_REGISTER_BLOCK_0,
    &&CPPC_RESTART_BLOCK_0};
  int cppc_next_jump_point = 0;
  MPI_Init( &argsc, &argsv );
  CPPC_Init( &argsc, &argsv );
  /* Conditional jump to next REC: code */
  if( CPPC_Jump_next() ) {
    const int jump_index =
      cppc_next_jump_point;
    cppc_next_jump_point++;
    goto * cppc_jump_points[ jump_index ];
  }

  MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
  MPI_Comm_size( MPI_COMM_WORLD, &worldsize );
  /* Check and obtain parameters */
  ... /* Not executed on restart */
  /* Initialize game boards */
  ... /* Not executed on restart */
CPPC_EXECUTE_BLOCK_0:
  /* Initialize communications and types */
  MPI_Cart_create( ..., &gridComm );
  ...
  /* Find TIDs for adjacent processes
   in the cartesian grid */
  ...
  /* Conditional jump to next REC */
  ...

CPPC_REGISTER_BLOCK_0:
  CPPC_Register( 0, local_size, 1, CPPC_INT,
    ..., CPPC_STATIC );
  local_board =
    CPPC_Register( 0, local_board, local_size,
      CPPC_CHAR, ..., CPPC_DYNAMIC );
  CPPC_Register( 0, mytid, 1, CPPC_INT, ...,
    CPPC_STATIC );
  CPPC_Register( 0, worldsize, 1, CPPC_INT,
    ..., CPPC_STATIC );
  /* Conditional jump to next REC */
  ...

  while( 1 ) {
CPPC_RESTART_BLOCK_0:
    CPPC_Do_checkpoint( 0 );
    /* Execution is resumed
     from this point */
    /* Original code for the loop here */
    ...
    if( option == 'q' ) {
      CPPC_Shutdown();
      MPI_Finalize();
      exit( 0 );
    }
  }
}
```

(b) Code after CPPC precompiling

Fig. 2. Example of fault-tolerance insertion through CPPC

writer, binary writer with ZLib compression and HDF-5 writer. Variable level checkpointing achieves very important size reductions on some applications when compared to data segment level, like in CG where this reduction reaches 60%.

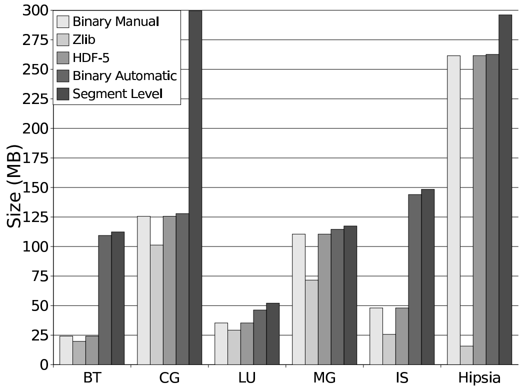

Fig. 3. Generated per-node file sizes

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 reason for delaying conversions until restart is that they may not even be necessary (e.g. restart on the same environment), thus reducing overhead of checkpoint operation.

The small reduction in file sizes obtained by the variable-level approach working with Hipsia is caused by the special characteristics of this application. As it is a Fortran 77 application, statically allocated arrays are oversized to fit a maximum problem size. As a result, an important amount of memory is wasted when executing this application. This is the reason why enabling file compression reduces the size of the state file in a significant percentage as can be seen in Figure 3.

The results tagged as “Binary Automatic” were generated by enabling the automatic variable analysis of the CPPC precompiler. Sizes obtained using manual analysis (which are the optimal ones) are very close to those obtained using automatic analysis, except for BT and IS applications. This is due to the fact that unnecessary array sections are registered because of the conservative approach of the precompiler, as explained in Section 4.

6.2 Checkpoint overhead

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, as shown in Figure 4. It shows raw dumping times, not the real contribution of dumping times to execution time, which is reduced by using multithreaded dumping. The HDF-5 writer shows better performance than the binary one, as no data conversion is being made in checkpoint operation; moreover, 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).

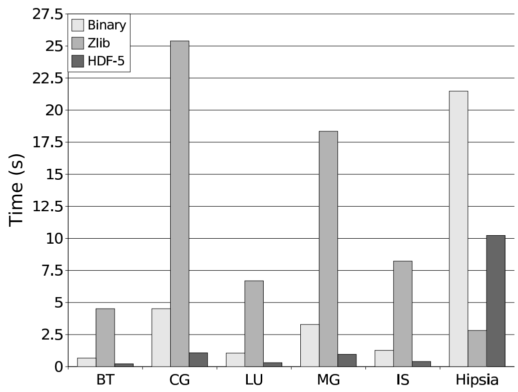

Fig. 4. File generation overhead

Using compression usually increases overall dumping time. Therefore, ZLib compression should be used only when the compression rate is high enough as to greatly reduce writing time, so that the overall dumping time is reduced. An example of such a situation can be seen when executing Hipsia. The obtained compression rate is so high (93.98%) that using ZLib significantly improves total dumping time, specially when working with slow disks. Using compression is also useful, for instance, if there are problems with disk quotas.

To reduce the overhead introduced by file generation, multithreaded state dumping has been implemented. Table 1 details the original execution times (without checkpointing) and the overhead introduced by the generation of a single checkpoint file using the writing strategy which achieves the best performance. Good results are obtained with those applications with higher runtimes. MG and IS have such a short execution time that creating a state file generates a high relative overhead (although this overhead is in the range of only seconds). Note that checkpoint overheads include, besides state dumping, the remaining CPPC operations: initialization, register management and finalization.

Table 1. Multithreaded checkpoint overheads (in seconds)

	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%
Hipsia	282.08	2.82	1.00%

6.3 Restart overhead

Execution overhead has been measured assuming that there are no failures during the execution. If failures occurred, restart time should be also considered in total execution time. Restart times for the previous applications have been measured and split into its three phases: negotiation, file reading and effective recovery. Results can be seen in Figure 5. Negotiation time and file reading time greatly depend on file sizes as they must be checked 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 for 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.

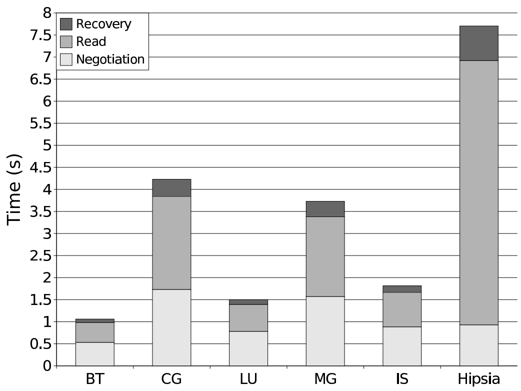

Fig. 5. Restart times for test applications

7 Related work

There are currently several solutions that deal with checkpointing for parallel applications. The main difference between these solutions and CPPC is portability, whether code portability or checkpoint file portability.

MPICH-GF [24] and MPICH-V2 [3] are implemented as MPICH 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 [19] implementation for Intel Paragon architectures. File portability is thus a nonsense 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 extensions [2] achieve to perform portable checkpointing, as they use the Dome environment [1] 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 at 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 [21] was developed for the Condor system [23]. It works at segment level, thus discarding file portability. It uses process coordination to achieve global consistency, thus being a non-scalable approach.

In [6] a variable-level approach is used. The programmer is responsible for performing relevant variable analysis, and inserting fault tolerant calls and flow control mechanisms into the original code. Performance overhead is reduced by restarting only faulted processes and using diskless checkpointing, based on a floating-point coding approach, which introduces round-off errors upon data recovery.

C^3 [20] 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 [4] 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 trait due to the inherent heterogeneity 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 (GridCPR) [10]. The checkpointing approach based on statically solving consistency issues improves the scalability of the tool. Automated tasks have been implemented into the CPPC precompiler in order to alleviate user effort, specially insertion of control flow code and data flow analysis for variable registration, which are the most time-consuming and error-prone tasks to be performed manually. All these features have been experimentally tested and encouraging results have been obtained.

Unfortunately, a comparative performance analysis between CPPC and the approaches mentioned in the related work section that were compatible with our experimental setup has not been possible, due to lack of implementations available on the web. An evaluation version of CPPC can be downloaded at `http://www.des.udc.es/~juan/cppc-0.3.tgz`

We are currently working on techniques for automatically or semi-automatically extracting information about safe points and code blocks that must be re-executed at restart, which would result in an almost completely transparent approach to checkpointing.

References

1. A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed Object Migration Environment. Technical Report CS-94-153, Carnegie-Mellon University, 1994.
2. A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.
3. 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.
4. K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.
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)*, pages 1–11, San Jose, CA, Nov. 1997.
6. Z. Chen, G.E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault Tolerant High Performance Computing by a Coding Approach. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 213–223, Chicago, IL, June 2005. ACM.
7. E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.
8. J. Gailly and M. Adler. ZLib Home Page. http://www.gzip.org/zlib/.

9. M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, (223):120–123, October, 1970.
10. Global Grid Forum. Global Grid Forum Grid Checkpoint Recovery Working Group. http://gridcpr.psc.edu/GGF/.
11. P. González, T.F. Pena, and J.C. Cabaleiro. Dual BEM for crack growth analysis on distributed-memory multiprocessors. *Advances in Engineering Software*, 31(12):921–927, Nov. 2000.
12. J-M. Hélary, R.H.B. Netzer, and M. Raynal. Consistency Issues in Distributed Checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274–281, March/Apr. 1999.
13. S-I. Lee, T.A. Johnson, and R. Eigenmann. Cetus – An extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 539–553, College Station, TX, 2003. Springer.
14. K. Li, J.F. Naughton, and J.S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, Aug. 1994.
15. S.S. Muchnick. *Advanced compiler design and implementation*. Morgan Kauffman, 1997. Chapter 14.
16. National Aeronautics and Space Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.
17. National Center for Supercomputing Applications. HDF-5: File Format Specification. http://hdf.ncsa.uiuc.edu/HDF5/doc/.
18. J.S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.
19. J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, New Orleans, LA, Jan. 1995.
20. M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, page 38, Pittsburgh, PA, Nov. 2004. IEEE Computer Society.
21. G. Stelnner. CoCheck: checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 526–531, Honolulu, HI, 1996, IEEE Press.
22. V. Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(5):22–32, September/October 1998.
23. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2):323–356, 2005.
24. N. Woo, H.S. Jung, H.Y. Yeom, T. Park, and H. Park. MPICH-GF: transparent checkpointing and rollback-recovery for Grid-enables processes. *IEICE Transactions on Information and Systems*, E87-D(7):1820–1828, 2004.