Journal of Universal Computer Science, vol. 20, no. 9 (2014), 1352-1372
submitted: 11/3/14, accepted: 16/6/14, appeared: 1/9/14

Extending an Application-Level Checkpointing Tool to Provide Fault Tolerance Support to OpenMP Applications

Nuria Losada, María J. Martín, Gabriel Rodríguez, Patricia González
(Computer Architecture Group, University of A Coruña, Spain
{nuria.losada, mariam, grodriguez, patricia.gonzalez}@udc.es)

Abstract: Despite the increasing popularity of shared-memory systems, there is a lack of tools for providing fault tolerance support to shared-memory applications. CPPC (ComPiler for Portable Checkpointing) is an application-level checkpointing tool focused on the insertion of fault tolerance into long-running MPI applications. This paper presents an extension to CPPC to allow the checkpointing of OpenMP applications. The proposed solution maintains the main characteristics of CPPC: portability and reduced checkpoint file size. The performance of the proposal is evaluated using the OpenMP NAS Parallel Benchmarks showing that most of the applications present small checkpoint overheads.
Key Words: parallel programming, OpenMP, fault tolerance, checkpointing
Category: D.1.3, D.4.5

1 Introduction

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 the execution of an application remains well under the mean time to failure of the underlying hardware. However that is not always possible on applications with long runs, where users and programmers need to make use of fault tolerance techniques to ensure that not all computation done is lost on machine failures.

In the early 2000s computer architecture trends switched to multicore scaling as a response to various architectural challenges that severely diminished the gains of further frequency scaling. This approach has enabled processors to take advantage of increasing transistor counts, according to Moore's Law, for the last decade. As a result, shared memory can be found in most current computational systems, ranging from desktops to large supercomputers. [OpenMP] is the de-facto standard for parallel programming on shared-memory systems. There exist many applications written in either pure OpenMP or a combination of MPI and OpenMP. However, most of the research on fault tolerance for parallel applications has focused on the message-passing model and the distributed memory systems [Beguelin et al. 1994, Bouteiller et al. 2003, Chen et al. 1997, Stellner 1996, Sunil et al. 2003, Woo et al. 2004], and there is a lack of tools for shared-memory applications.

Checkpointing is a widely used fault tolerance technique, in which the computation state is saved periodically to disk, allowing the recovery of the application when a failure occurs. CPPC (ComPiler for Portable Checkpointing) [Rodríguez 2008] is a portable and transparent checkpointing infrastructure for MPI [Walker and Dongarra 1996] parallel applications. It is made up of a runtime library containing checkpoint-support routines and a compiler that automates the use of the library. In this work the CPPC library is modified and extended to provide fault tolerance support to OpenMP applications. This extension preserves the main characteristics of the CPPC tool: portability and reduced checkpoint size. The solution proposed will allow to insert fault tolerance into both OpenMP and hybrid OpenMP/MPI applications.

This paper is structured as follows. [Section 2] covers Related Work. [Section 3] introduces the CPPC tool. [Section 4] describes how CPPC is modified and extended to cope with OpenMP applications. [Section 5] presents the experimental results using the NAS Parallel Benchmarks. Finally, [Section 6] concludes this paper.

2 Related work

Fault tolerance for parallel applications is a very active research topic with a large number of approaches published in the last two decades. In this section we will focus on proposals that, like the one proposed in this work, address fault tolerance for shared memory applications.

There exist in the literature several solutions based on checkpointing. The main difference with the one proposed in this paper is portability, whether code portability (allowing its use on different architectures) or checkpoint files portability (allowing to restart on different machines).

ReVive [Prvulovic et al. 2002] and SafetyNet [Sorin et al. 2002] are hardware solutions. They perform checkpointing and logging to achieve fault tolerance on shared memory multiprocessors relying on specific hardware. They present a high efficiency, but they are inherently platform dependent.

[Dieter and Lumpp 1999] describe a checkpointing library for multithreaded programs that uses the POSIX threads library provided by Solaris 2. Fault tolerance is achieved by the instrumentation of the original source code. During the execution, the library sends Unix signals to all the threads to synchronize them at a checkpoint. Some parts of the library are platform dependent, so again this is a non-portable solution.

DMTCP [Ansel et al. 2009] is a transparent user-level checkpointing package for distributed multithreaded applications, including hybrid OpenMP/MPI applications. It uses a coordinated checkpointing method where all processes and threads cluster-wide need to be simultaneously suspended during checkpointing. DMTCP does not depend on any specific message-passing library, nor on kernel modification. It is compatible with Linux distributions running on x86, x86_64, or ARMv7 architectures. However, since it stores and recovers the entire user space, it does not allow for restart in environments with non-portable user-space structures (e.g. heterogeneous computing environments with user-space communication drivers).

C^3 [Bronevetsky et al. 2004, Bronevetsky et al. 2006] is a checkpoint compiler that follows an application-level approach, achieving fault tolerance through the instrumentation of the original application. C^3 intercepts all calls to the OpenMP library and reimplements some of the OpenMP functionalities. It can be used on different platforms, but the generated checkpoint files are not portable, as it enforces data to be recovered at the same virtual address as in the original execution to achieve pointer consistency.

[Fu and Ding 2010], as well as [Tahan and Shawky 2012], follow an approach based on redundancy instead of checkpointing. These approaches avoid the overhead of I/O transfer present in checkpoint techniques. However, they can only detect the errors but not correct them if multiple copies of the computation fail.

3 CPPC overview

CPPC is an open-source checkpointing tool for MPI applications available at `http://cppc.des.udc.es` under GNU general public license (GPL). It is implemented at the application level, and, thus, it is independent of the MPI implementation, the operating system, and any higher-level framework used.

CPPC reduces the amount of data to be saved by storing user variables and using a liveness analysis to save only those variables that are indispensable for the application recovery. Besides, CPPC applies another snapshot size reduction technique, zero-blocks exclusion [Cores et al. 2013], which consists in avoiding the storage of memory blocks that contain only zeros.

Generated state files are portable, allowing the execution to restart on different architectures and/or operating systems. Portability is achieved by using a portable storage format and by avoiding the inclusion of architecture-dependent state in checkpoint files. CPPC uses [HDF5] (Hierarchical Data Format 5), a data format and associated library for the portable transfer of graphical and numerical data between computers. The architecture-dependent state is recovered through the re-execution of the code responsible for creating such state in the original execution.

CPPC follows a distributed and non-coordinated approach, described in [Rodríguez et al. 2009], where checkpoints are taken independently by each process and consistency is guaranteed by locating checkpoints at *safe points*, where there are no pending communications. During the restart execution, the processes perform a negotiation to achieve an agreement about which subset of checkpoint files will be used for the application recovery.

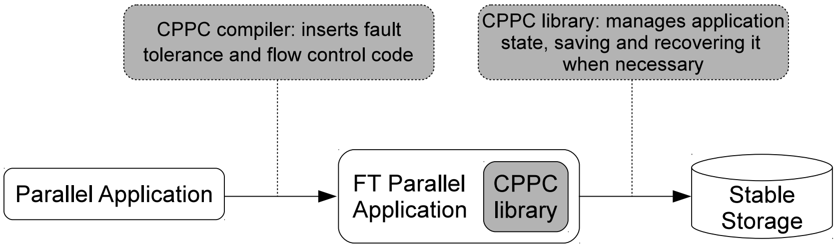

Figure 1: *CPPC global flow.*

CPPC appears to the final user as a compiler tool and a runtime library. At compile time the CPPC source-to-source compiler automatically transforms a parallel code into an equivalent fault-tolerant version with calls to the CPPC library, see [Fig. 1] and [Fig. 2], to perform the following actions:

– **Configuration and initialization:** at the beginning of the application, `CPPC_Init_configuration()` and `CPPC_Init_state()` perform the configuration and initialization of the necessary data structures for the library management.

– **Registration of variables:** using the `CPPC_Register()` routine, variables necessary for the successful recovery of the application are explicitly marked for inclusion in checkpoint files. During restart, this routine will also perform the recovery of the values from the checkpoint files to their proper memory location. A `CPPC_Unregister()` routine is also provided, allowing to remove obsolete registered variables that do not have to be included in future checkpoint files.

– **Checkpoint:** the `CPPC_Do_checkpoint()` routine is added at selected safe points inside the most computationally expensive loops of the application. A checkpoint file will be generated every N calls to this function, being N user-defined. At restart time this routine checks restart completion.

– **Conditional jumps**: flow control code is added (`CPPC_Jump_next()` routine and labels) to re-execute selected state-recovering statements during the restart.

– **Shutdown:** the `CPPC_Shutdown()` routine is added at the end of the application to ensure consistent system shutdown.

The restart phase has three fundamental parts: finding the recovery line (the subset of checkpoint files that will be used), reading the checkpoint data into

```
int main()
{
   CPPC_Init_configuration();
   CPPC_Init_state();
   if (CPPC_Jump_next())
      goto REGISTER_BLOCK_1;

   [...]

   REGISTER_BLOCK_1:
   <CPPC_Register() block>
   if (CPPC_Jump_next())
      goto RESTART_BLOCK_1;

   [...]

   RESTART_BLOCK_1:
   compute();

   [...]

   <CPPC_Unregister() block>
   [...]
   CPPC_Shutdown();
}
```

```
void compute()
{
   if (CPPC_Jump_next())
      goto REGISTER_BLOCK_2;

   [...]

   REGISTER_BLOCK_2:
   <CPPC_Register() block>
   if (CPPC_Jump_next())
      goto RESTART_BLOCK_2;

   [...]

   for(i=0;i<niters;i++)
   {
      RESTART_BLOCK_2:
      CPPC_Do_Checkpoint();

      [...]
   }

   <CPPC_Unregister() block>
   [...]
}
```

Figure 2: *Instrumentation schema.*

memory, and recovering the application state. The first two steps are encapsulated inside the `CPPC_Init_state()` call. In order to achieve complete state reconstruction it is necessary to recover not only data pertaining to user variables, but also state created in the original execution that cannot be portably stored into a checkpoint file. The application stack is an example of such non-portable state. CPPC uses selective code re-execution to achieve complete application state recovery. In this way, non-portable state is recovered by re-executing the instructions that created such state in the original run. In particular, the application stack is recovered by recreating the sequence of original function calls that built the stack up to the checkpoint location. In order to do this, the CPPC compiler instruments the original code with jumps between the appropriate sections of code. These jumps are only taken during an application restart. In the example in [Fig. 2] a jump is taken immediately after `CPPC_Init_state()` to skip the code up until the `CPPC_Register()` block. After recovering the values of user variables, another jump is taken, reaching the function where the checkpoint is located. If the `compute()` call receive any parameters, their values would be stored by the library in the checkpoint file so that during a restart an identical call would be reissued. The same process is followed inside the `compute()` function. Once all values of the user variables are recovered by the code labeled `REGISTER_BLOCK_2`, particularly that of the loop index variable `i`, another jump is taken, moving control into the loop to execute the `CPPC_Do_checkpoint()` function. Here, CPPC checks that control has reached the original checkpoint location and the restart phase is deactivated. Note that, by skipping the loop header through the jump, the recovered `i` value is preserved. For an in-depth description of the design and implementation of CPPC and its restart protocol, the reader is referred to [Rodríguez et al. 2010, Rodríguez et al. 2011].

4 Using CPPC on OpenMP applications

OpenMP is an API (Application Programming Interface) for shared-memory parallelism in C, Fortran and C++. It is portable across shared-memory architectures from different vendors. The specification defines a collection of compiler directives, library routines and environment variables that implements multithreading with the fork-join model, based on the creation and destruction of threads.

The following subsections describe how CPPC is modified and extended to cope with OpenMP applications.

4.1 Configuration, initialization and shutdown

In the MPI programming model the user specifies the number of processes at the beginning of the program, and typically this number remains constant throughout the execution. In contrast, OpenMP provides a dynamic form of parallelism in which threads are created and destroyed during the program execution. In an OpenMP program, when a thread encounters a `parallel` directive a team of threads is created and all of them execute concurrently the parallel region located inside the directive. The thread that executes the parallel directive becomes the master thread of the new team.

From the CPPC perspective, this involves the need to create and initialize data structures for the management of the master thread and each new thread created along the program. To avoid overestimating the number of threads, each new thread is responsible for the initialization of its private structures and for the inheritance of the necessary configuration parameters from the master thread. Thus, the configuration of the library continues to be at the beginning of the program, but now, new initialization and shutdown blocks are added at the beginning and at the end of each parallel region, respectively. The process is represented in [Fig. 3].

4.2 Registration of variables

The live variable analysis provided by the CPPC compiler is reused to identify those variable values that need to be saved for the correct restart of the exe-

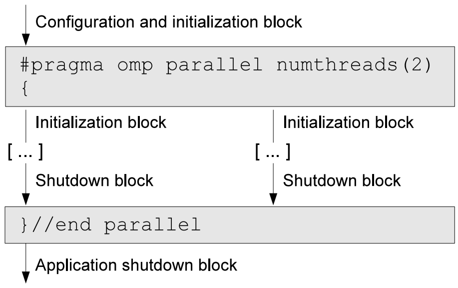

Figure 3: *Parallel regions management.*

cution. Depending on the considered application, the use of this technique can significantly reduce checkpoint file sizes [Cores et al. 2013].

In MPI each process has its own local variables so that in CPPC-instrumented MPI-codes each process saves its local computational state in its own checkpoint file. In contrast, OpenMP variables can be private or shared. For private variables, each thread has its own memory copy. In the case of shared variables, there is only one copy accessible by all threads.

CPPC has been extended to distinguish between private and shared variables. Each thread saves its private variables into its own checkpoint file. Shared variables, on the other hand, will only be included in the checkpoint file of the master thread. Private variables are registered using the original `CPPC_Register()` routine. For the registration of shared variables, the `CPPC_Register_shared()` routine has been added to the library. This routine will be called by all the threads but only the master thread will include the values of the shared variables into its state files. Its execution by all threads is mandatory as, during restart, synchronizations are performed to avoid accesses to a shared variable before the master thread recovers its value.

4.3 Checkpoint file dumping and consistency protocol

The `CPPC_Do_checkpoint()` routine is inserted inside the most computationally expensive loops of the OpenMP application using the heuristic analysis provided by the CPPC compiler [Rodríguez et al. 2009] to locate them. As mentioned in [Section 3], checkpoints are stored using [HDF5], which enables the restart on different architectures, and applies a file size reduction technique, the zero-blocks exclusion [Cores et al. 2013].

When there are not any shared variables registered a non-coordinated checkpointing is performed. Otherwise, a coordinated approach is needed to ensure that both private and shared state are consistently stored to the checkpoint files, allowing the recovery of a consistent global state.

The coordinated protocol followed is similar to the one used in C^3, described in [Bronevetsky et al. 2004, Bronevetsky et al. 2006]: all threads are forced to generate a checkpoint file at the same time. For this purpose, a checkpoint flag is used to indicate when a checkpoint is in process, and OpenMP barriers are inserted before and after the state dumping step to ensure that no thread can continue its execution until all threads have reached that point. Additionally, OpenMP barriers contain implicit flushes of shared variables, guaranteeing a consistent view of memory.

Note that the inclusion of barriers in the checkpoint operation can interfere with other barriers present in the application, including implicit barriers present in some OpenMP directives, such as the `for` directive (detailed in [Section 4.5]). Those barriers are thus replaced with a call to `CPPC_Barrier()`, a new library routine that includes an OpenMP barrier and a conditional call to the checkpoint routine. In this way, a checkpoint file is generated inside `CPPC_Barrier()` when the checkpoint flag is activated, avoiding deadlocks.

4.4 Parallel regions

If a checkpoint call is inserted inside a parallel region then it has to be guaranteed that the same number of threads as in the original execution will be created during the restart. Otherwise, the restart will not be successful. If more threads are created, some of them will try to read a non existent checkpoint file. If fewer threads are created, not all the application state will be recovered.

For this reason, parallel directives are instrumented to include their original number of threads in the checkpoint files. During the restart, the dynamic adjustment of the number of threads is disabled and the `omp_set_num_threads()` routine is called to force the creation of exactly the same number of threads as in the original execution. CPPC will abort the restart if the number of threads requested is greater than the limit specified by the `thread-limit-var` OpenMP internal control variable.

4.5 Parallel loops

When the OpenMP `for` directive is applied to a loop inside a parallel region, its iteration space is divided among the threads according to the `schedule(type, chunk_size)` clause. The scheduling *type* can be `static`, `dynamic` or `guided`, and the *chunk_size* value is used to generate the subsets of iterations.

As shown in [Fig. 4], when checkpoints are taken inside a parallel loop, the iteration space will be formed by chunks where all the iterations are completely processed, chunks where some iterations have been processed, and chunks where all the iterations are unprocessed.

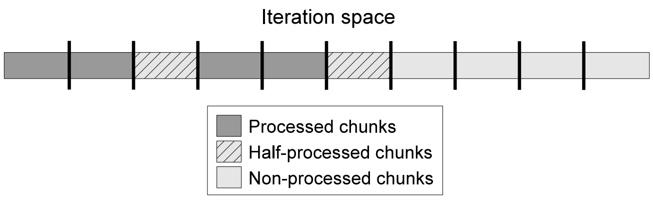

Figure 4: *Representation of the state of a parallel loop.*

Thus, some information has to be included into the checkpoint files in order to skip, during the restart, those iterations that were already processed during the original execution. The following subsections detail how the restart process is managed for each of the scheduling types.

4.5.1 `Static` scheduling

With the `static` scheduling the iteration space is divided into chunks of size *chunk_size* distributed among threads in a round-robin fashion. This scheduling is deterministic: different executions will produce the same chunks and the same assignment to threads.

The solution is simple in this case, as each thread only needs to include in its checkpoint file the value of the furthest iteration index it has processed. During the restart execution, each thread will be assigned the same chunks and it will skip all iterations until it gets to the iteration recovered from its checkpoint file. At that moment the restart process finishes, and the subsequently assigned iterations are executed normally.

4.5.2 `Dynamic` scheduling

Using `dynamic` scheduling the iteration space is also divided into chunks of size *chunk_size*, but now they are assigned to threads in a dynamic and non-deterministic way. This assignment can cause that, during the restart, the same thread gets more than one half-processed chunk, having to alternate between skipping already-executed iterations and executing pending iterations to recover the application.

To manage the restart process, the same information as in the `static` scheduling is included in the checkpoint files, but now the information is shared among the threads. [Fig. 5] shows an example for a 20 iteration loop executed using 4 threads, where $X[j]$ $(0 \leq j \leq T-1$, being T the number of threads) is the shared array that stores the index of the furthest iteration processed by each thread.

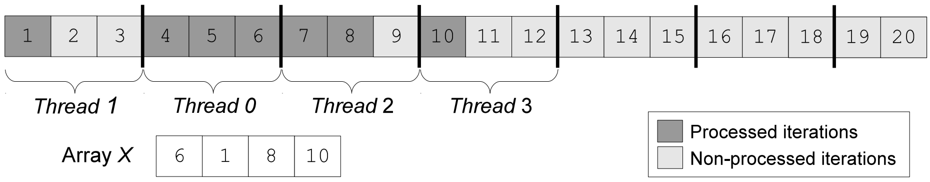

Figure 5: *Processing state upon checkpoint creation in a dynamic scheduling.*

Using this information, at restart time each thread will be able to determine if the assigned chunk belongs to a processed, half-processed, or non-processed chunk and take the appropriate actions.

Without loss of generality, lets assume a forward loop. Let $\{i^t_0, i^t_1, \ldots, i^t_{k-1}\}$ be the set of indexes of the iterations of the chunk assigned to thread t, being k the chunk size:

– If $i^t_0 > X[j]\ \forall j$, a non-processed chunk is assigned. Restart process is finished and the execution continues normally as it is guaranteed that no more processed or half-processed chunks will be assigned.

– If $\exists\ j : i^t_0 < X[j] < i^t_{k-1}$, a half-processed chunk is assigned. Iterations with index in the range $[i^t_0, X[j]]$ will be skipped and iterations with index in the range $(X[j], i^t_{k-1}]$ will be computed.

– If none of the two previous conditions is satisfied, a processed chunk is assigned. All iterations in this chunk will be skipped.

For backwards loops the analysis would be similar but reversing the direction of the inequalities.

4.5.3 `Guided` scheduling

In a `guided` scheduling, the chunk assignment is performed as in the `dynamic` scheduling, but now the chunk size is dependent on the particular OpenMP implementation used. From the CPPC library perspective, both the chunk size and the chunk assignment are considered unknown and a new approach is used to manage this scheduling type.

In order to calculate at restart time the set of processed and pending iterations, the processed iterations are grouped in subsets of consecutive iterations and represented with integer pairs as (f_j, l_j), being f_j the index of the first iteration and l_j the index of the last iteration of each processed subset j. These

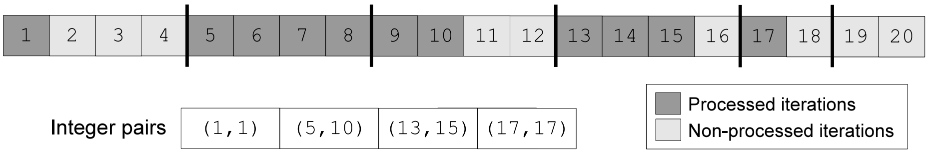

Figure 6: *Processing state upon checkpoint creation in a guided scheduling.*

pairs are updated by the threads as they execute the iterations and are included in the checkpoint files as shared information. [Fig. 6] shows the processing state of a 20 iteration loop executed with guided scheduling using 4 threads, and the representation of this state with a list of 4 pairs.

Without loss of generality, lets assume a forward loop. Let i_t be the index of the current iteration assigned to thread t:

- If $\nexists\ j : i_t < l_j$, change to normal execution as the recovery process has finished.
- If $\exists\ j : f_j \leq i_t \leq l_j$, skip all the iterations until $i_t > l_j$.
- If none of the two previous conditions is satisfied: compute all iterations until getting an iteration with index greater or equal to $f_k = \min_j\{f_j : i_t < f_j\}$.

For backwards loops the analysis would be similar but reversing the direction of the inequalities.

4.6 Reduction operations

OpenMP allows the specification of reduction operations in some directives, such as `parallel` or `for`, using the `reduction(operator:list)` clause. For each variable in the list, OpenMP creates a private copy for each thread, initialized appropriately for the operator. All references to the variable in the reduction region affect the private copy. After the reduction region, the original variable will be updated with the private copies using the specified operator.

4.6.1 `Parallel` directive

For managing a reduction operation in a `parallel` directive the following steps are taken:

1. The shared reduction variable is registered before the parallel region using the `CPPC_Register_shared()` routine. This will make possible to recover its initial value during the restart execution.

2. At the beginning of the parallel region, each thread will register its private copy using the `CPPC_Register()` routine. This will include into the checkpoint files the values that correspond to the work performed by each thread.

During the restart the register routines will perform the recovery of the shared and the private copies of the reduction variable from the checkpoint files.

4.6.2 `For` directive

The management of reduction operations in a `for` directive is more complex, since it is not guaranteed that all threads will execute the body of the loop. For example, if the *chunk_size* specified in a `for` directive produces fewer iteration subsets than threads executing the directive, some threads will not do any work at all. This involves that one thread could execute the body of the directive during the original execution, but not during the restart.

Thus, during a restart the shared and private copies of the reduction variable are recovered before the `for` directive, and the initial value of the shared reduction variable is updated to the result of a partial reduction. The partial reduction will use the initial value of the shared reduction variable and each private copy. In this way, the shared reduction variable will store the result of the job performed in the original execution. This approach avoids the loss of computed job and guarantees the correctness of the result even with multiple restart executions.

New routines have been added to the library to perform these operations:

– `CPPC_Register_reduction()`: called inside the `for` region to register the private copies of the reduction variable.

– `CPPC_Unregister_reduction()`: called at the end of the `for` region to remove the registrations associated to a reduction.

– `CPPC_Init_reduction()`: called before the `for` region to register the shared reduction variable during a normal execution and to recover the shared and private copies of the reduction variable and perform the partial reduction of their values during a restart.

5 Experimental results

The following subsections assess the overhead of the solution, distinguishing between the overhead of the checkpoint operation and the restart procedure.

The application testbed used, summarized in [Tab. 1], was comprised of the ten applications of the NAS Parallel Benchmarks (NPB) [NAS NPB] v3.3 for

	Description	Number of threads				
		1	**2**	**4**	**8**	**16**
FT	Fourier Transform	280.72	140.44	72.03	38.37	22.18
MG	MultiGrid	61.17	30.21	16.11	9.38	7.76
IS	Integer Sort	36.74	19.42	9.87	5.10	2.73
CG	Conjugate Gradient	304.99	142.83	75.19	41.37	24.33
BT	Block Tri-diagonal solver	1048.07	525.91	270.30	144.62	87.00
SP	Scalar Penta-diagonal solver	643.30	320.74	168.79	103.33	156.69
UA	Unstructured Adaptive mesh	838.41	428.33	234.95	138.49	94.84
DC	Data Cube	582.78	313.45	185.05	117.72	78.40
EP	Embarrasingly Parallel					
	Static scheduling	277.98	139.07	71.04	36.27	19.34
	Dynamic scheduling	277.65	138.83	70.28	36.19	19.27
	Guided scheduling	278.65	139.53	70.62	36.41	19.39
LU	Lower-Upper Gauss-Seidel					
	Static scheduling	762.30	354.58	183.51	98.48	58.40
	Dynamic scheduling	764.99	354.27	183.61	99.05	58.60
	Guided scheduling	764.96	354.36	183.59	99.14	58.50

Table 1: *Original runtimes (s) for the testbed benchmarks.*

OpenMP. These are well-known and widespread applications that provide a de-facto test suite. Out of the NPB suite, the DC benchmark was run using class B, while the rest were run using class C.

The CPPC compiler automatically determines the checkpoint location inside the most computationally expensive loops of the application. In all the NPB applications, except EP, DC and LU, the checkpoint is placed inside a sequential main loop outside the parallel region. In DC the checkpoint is located in a sequential loop inside a parallel region. In EP and LU the checkpoint is placed in a parallel loop. Furthermore, EP presents multiple reduction operations in the `for` directive. Both EP and LU will be used to compare, in terms of efficiency, the management of the different scheduling types (static, dynamic and guided).

Runtime tests were performed in the Pluton cluster, hosted by the University of A Coruña. Each node consists of two Intel Xeon E5-2660 Sandy Bridge-EP 2.2 GHz processors, with 8 cores per processor and 64 GB of RAM. State files were stored in an NFS mounted directory, using a Gigabit Ethernet network.

The CPPC version used was 0.8.1, working along with HDF5 v1.8.11 and GCC v4.4.7.

Each result in this section was obtained after performing 10 experiments.

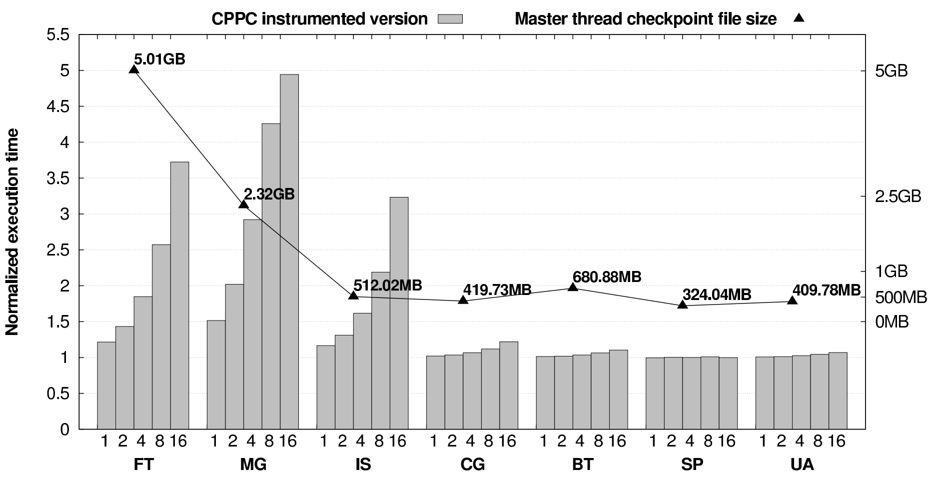

Figure 7: *Checkpoint overhead: checkpoint located outside a parallel region.*

Average runtimes are reported. [Tab. 1] shows the original runtimes for all the NPB applications, that is, the runtime of the non-fault-tolerant version.

5.1 Checkpoint overhead

The overhead of a fault tolerant solution based on checkpointing depends deeply on the overhead introduced by the checkpoint file dumping. This overhead is tightly tied to the size of the data to be dumped. When the fault tolerant solution is applied to a parallel application, the consistency protocol between threads may also impact the overhead of the checkpoint operation, due to the synchronizations introduced by the protocol.

[Fig. 7] and [Fig. 8] show the runtimes of the CPPC instrumented versions. As one checkpoint file is generated for each N calls to `CPPC_Do_checkpoint()`, in these experiments the dumping frequency (N) is set to the appropriate value for each application (using the configuration parameters of CPPC) so that a state file is generated when the 75% of the computation has been completed. For legibility reasons, runtimes are normalized with respect to the original execution of the application using the same number of threads. The master thread checkpoint file sizes are also shown in the figure. In DC, EP and LU, in which the checkpoint is placed inside a parallel region, the rest of the threads also generate checkpoint files, but the size of these files is always smaller than 0.13 MB/thread.

Most of these applications present small relative checkpoint overheads. However, as can be seen, three of the ten applications, FT, MG and IS, present larger

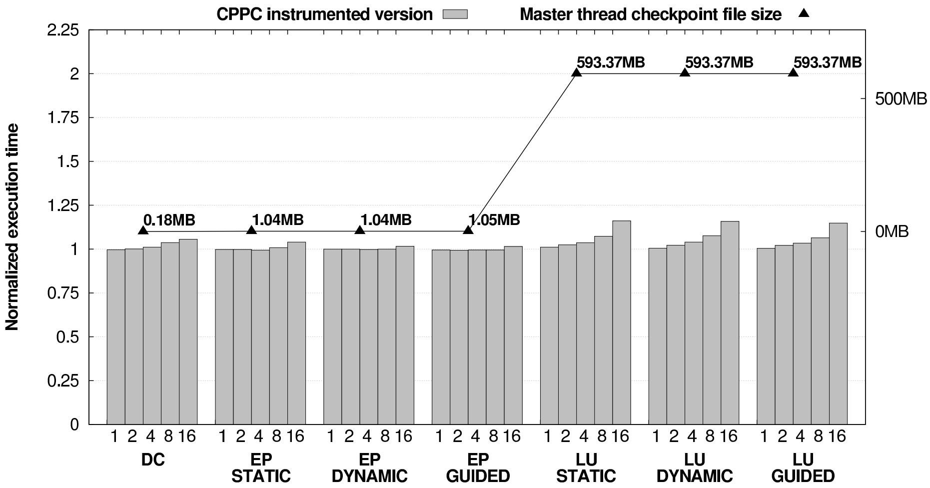

Figure 8: *Checkpoint overhead: checkpoint located inside a parallel region.*

overheads. As mentioned above, the overhead heavily depends on the state file size. These sizes in FT and MG exceed, respectively, 5 GB and 2 GB, and therefore, the observed checkpoint overhead is large for both of them. Note that the overhead would be larger if the live variable analysis and the zero-blocks exclusion [Cores et al. 2013] were not used (for example, the checkpoint file size in MG would be 3.32 GB instead of 2.32 GB). The impact of this overhead obviously depends on the original application runtime. Since the original runtime for MG is lower than for FT, its checkpoint overhead, in percentage, is significantly larger. The same reasoning applies to the results observed for IS, the high impact on the checkpoint overhead is due to the small original runtime of the application.

Although, as can be seen, the relative overhead increases with the number of threads, the absolute overhead keeps practically invariable. For those benchmarks with the checkpoint outside a parallel region, the absolute overhead remains constant as the state that needs to be saved (the shared state) does not vary with the number of threads and coordination is not required. For those with the checkpoint inside a parallel region, as the private variables represent only a very small percentage of the data to be checkpointed and the coordination cost is, in general, irrelevant (see [Subsection 5.1.1]), the absolute overhead does not differ significantly either. For example, MG presents an absolute overhead of approximately 31 seconds, independently of the number of threads. The large relative overhead of this application for sixteen threads is because of the small runtime of the original version with this number of threads. Inserting a checkpoint call in this case is equivalent to perform a checkpoint dumping of 2.32 GB every 7.76 seconds. In a real application the checkpoint frequency should be adjusted in a more realistic way to avoid this behavior.

With regard to the checkpoint overhead using different scheduling types in a parallelized loop (LU and EP benchmarks), results show that there are no relevant differences between the approaches used to manage the static, dynamic or guided scheduling.

5.1.1 Checkpoint file generation and consistency protocol overhead

As explained in [Section 4.3], if the checkpoint routine is called inside a parallel region, the consistency protocol adds synchronization operations before and after the checkpoint file generation. Hence, for DC, EP and LU it is interesting to study which part of the checkpoint overhead is due to the checkpoint file generation and which part is due to the consistency protocol.

[Tab. 2] shows the checkpoint file generation times (G) and the consistency protocol overheads (P), both in seconds. The checkpoint file generation time corresponds to the time elapsed during the largest state file generation, which corresponds to the generation of the master thread checkpoint file. The consistency protocol overhead is the time elapsed between the activation of the checkpoint flag by one thread, until all of them start the checkpoint generation once synchronization is reached.

The checkpoint file generation overheads are consistent with the state file sizes shown in [Fig. 7] and [Fig. 8]. Generation overheads slightly increase with the number of threads as each thread contributes with its private data.

For LU and EP the overhead introduced by the CPPC consistency protocol is negligible, as it adds less than 0.006 and 0.04 seconds respectively. DC, in contrast, presents a high consistency protocol overhead due to the particular characteristics of this application. As explained previously, in DC the checkpoint is located in a sequential loop inside a parallel region. It is a data intensive benchmark. The execution of each iteration involves multiple I/O operations, where threads compete for I/O bandwidth. Thus, in the original DC benchmark, as the number of threads increases, so does the variability of the iteration time, evolving its standard deviation from 1.69 seconds in the sequential version, to 4.56 seconds with sixteen threads. The difference between the fastest and the slowest thread is increased, which increments the consistency protocol overhead in turn.

5.2 Restart overhead

The execution overhead was studied in the previous section during a fault-free execution. If a failure occurred, the restart time overhead would play a funda-

		Number of threads				
		1	**2**	**4**	**8**	**16**
DC	**G**	0.0145	0.0365	0.0586	0.1210	0.2458
	P	0.0001	4.7133	5.6406	6.5198	8.3752
EP `static`	**G**	0.0201	0.0205	0.0306	0.0350	0.0566
	P	0.0001	0.0024	0.0033	0.0038	0.0042
EP `dynamic`	**G**	0.0202	0.0209	0.0254	0.0420	0.0567
	P	0.0001	0.0018	0.0029	0.0037	0.0043
EP `guided`	**G**	0.0245	0.0219	0.0290	0.0377	0.0608
	P	0.0001	0.0016	0.0035	0.0037	0.0044
LU `static`	**G**	7.0813	7.0971	7.0944	7.1069	7.1181
	P	0.0001	0.0001	0.0001	0.0002	0.0003
LU `dynamic`	**G**	7.0882	7.0945	7.0933	7.1090	7.1257
	P	0.0001	0.0001	0.0001	0.0001	0.0004
LU `guided`	**G**	7.1088	7.0973	7.0865	7.1002	7.1316
	P	0.0001	0.0001	0.0001	0.0002	0.0002

Table 2: *Checkpoint file generation (G) and consistency protocol (P) overhead.*

mental role in the global execution time.

Restart experiments were performed using the checkpoint files generated in the previous sections. Restart times have been split into two phases: reading (RT) and positioning (PT). The results are shown in [Tab. 3]. As can be seen, the reading is the most costly phase. It includes both the negotiation to decide the state files to be used, and the actual read of their contents. The positioning phase includes the execution of all the additional operations needed to recover the application, that is, the execution of each thread up to the point where the state file was generated.

Reading times, as expected, are consistent with the state file sizes (shown in [Fig. 7] and [Fig. 8]). The larger the checkpoint file is, the slower the reading phase becomes. The first part of the table presents the applications with the checkpoint located in a sequential region, and thus, a single checkpoint file is read. In those applications where the checkpoint is placed in a parallel region (DC, EP and LU), reading times slightly increase when more threads are involved in the negotiation process. The difference however is very small, a few hundredths of a second.

As for the positioning times, note that they are determined by two main factors: 1) the amount of data that must be copied to the proper memory location; and 2) the execution of the non-portable state recovery blocks. In all

			Number of threads				
			1	**2**	**4**	**8**	**16**
Sequential region checkpoint	**FT**	**RT**	47.90	48.06	47.95	48.06	47.92
		PT	2.30	2.30	2.31	2.31	2.31
	MG	**RT**	23.48	23.47	23.48	23.48	23.47
		PT	1.51	1.51	1.51	1.51	1.51
	IS	**RT**	4.76	4.76	4.90	4.76	4.76
		PT	0.36	0.29	0.26	0.25	0.25
	CG	**RT**	3.92	3.92	3.92	3.92	3.92
		PT	0.20	0.20	0.20	0.20	0.20
	BT	**RT**	6.60	6.59	6.60	6.59	6.60
		PT	0.30	0.30	0.30	0.30	0.30
	SP	**RT**	3.23	3.23	3.25	3.23	3.23
		PT	0.15	0.15	0.15	0.15	0.15
	UA	**RT**	3.83	3.83	3.83	3.83	3.83
		PT	0.20	0.19	0.19	0.19	0.19
Parallel region checkpoint	**DC**	**RT**	0.01	0.02	0.04	0.06	0.13
		PT	0.01	0.02	0.05	0.05	0.06
	EP `static`	**RT**	0.02	0.02	0.02	0.03	0.04
		PT	0.01	0.02	0.02	0.03	0.04
	EP `dynamic`	**RT**	0.02	0.02	0.02	0.03	0.04
		PT	0.02	0.02	0.01	0.02	0.02
	EP `guided`	**RT**	0.02	0.02	0.02	0.03	0.04
		PT	0.02	0.02	0.02	0.03	0.03
	LU `static`	**RT**	6.32	6.33	6.33	6.35	6.36
		PT	0.26	0.27	0.26	0.27	0.27
	LU `dynamic`	**RT**	6.31	6.33	6.33	6.35	6.35
		PT	0.26	0.26	0.26	0.26	0.27
	LU `guided`	**RT**	6.30	6.33	6.32	6.33	6.33
		PT	0.26	0.26	0.26	0.26	0.26

Table 3: *Restart times (s): reading time (RT) and positioning time (PT).*

of the testbed applications the positioning times are consistent with the state file sizes (shown in [Fig. 7] and [Fig. 8]). Also, for most applications positioning times do not vary with the number of threads. The exception is IS, where the positioning times decrease with the number of threads due to the re-execution, during restart, of a dynamic memory allocation block executed in parallel.

Finally, positioning times in EP or LU do not show differences between the approaches used to manage the restart of each one of the different scheduling types (static, dynamic or guided).

6 Concluding remarks

This paper presents the extension of CPPC, a checkpointing tool for message-passing applications, to achieve fault-tolerance in OpenMP applications. The proposed solution applies an application-level checkpointing using a portable and coordinated approach.

Modifications have been performed to the CPPC library, allowing the management of the necessary data structures for each thread. Also, new functionalities have been added to the library in order to deal with OpenMP main features, such as the creation and destruction of threads; registration mechanisms for both private and shared variables; and a coordinated checkpointing to allow the consistent dumping of the shared state. Finally, support for parallelized loops with different scheduling types and for the reduction operations is added to the library. This first extension of the CPPC library is focused on loop-level parallelism. As experimental results show, this allows for the checkpointing of all the applications included in the OpenMP NAS Parallel Benchmarks. The management of task-level parallelism will be studied in future work.

Experimental results show the good performance of the solution. It presents a good scalability and the absolute overhead introduced is practically independent of the number of threads executing the application. Also, the restart overhead remains almost constant between executions with different number of threads, and it is mainly determined by the size of the data that needs to be recovered.

However, and in spite of the checkpoint file reduction techniques applied, most of the overhead introduced is due to the state dumping. Thus, optimizations are being considered to reduce it, such as the distribution of the shared state among the threads that execute the application.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiviness of Spain and FEDER funds of the EU (Project TIN2013-42148-P) and by the Galician Government (consolidation program of competitive reference groups GRC2013/055).

References

[Ansel et al. 2009] Ansel, J., Arya, K., Cooperman, G.: “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”; Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium; (IPDPS’09); IEEE, 2009.

[Beguelin et al. 1994] Beguelin, A., Seligman, E., Starkey, M.: "Dome: Distributed object migration environment"; Technical Report CMU-CS-94-153; Carnegie Mellon University (1994).

[Bouteiller et al. 2003] Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., Magniette, F.: "MPICH-V2: A Fault Tolerant MPI for Volatile Nodes Based on Pessimistic Sender Based Message Logging"; Proceedings of the 2003 ACM/IEEE Conference on Supercomputing; (SC'03); 25; ACM, 2003.

[Bronevetsky et al. 2004] Bronevetsky, G., Marques, D., Pingali, K., Szwed, P., Schulz, M.: "Application-level checkpointing for shared memory programs"; Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems; (ASPLOS'04); 235-247; ACM, 2004.

[Bronevetsky et al. 2006] Bronevetsky, G., Pingali, K., Stodghill, P.: "Experimental evaluation of application-level checkpointing for OpenMP programs"; Proceedings of the 20th Annual International Conference on Supercomputing; (ICS'06); 2-13; ACM, 2006.

[Chen et al. 1997] Chen, Y., Li, K., Plank, J.: "CLIP: A Checkpointing Tool for Message Passing Parallel Programs"; Proceedings of the 1997 ACM/IEEE Conference on Supercomputing; 33–33; IEEE Computer Society, 1997.

[Cores et al. 2013] Cores, I., Rodríguez, G., Martín, M., González, P., Osorio, R.: "Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes"; New Generation Computing; 31 (2013), 3, 163-185.

[Dieter and Lumpp 1999] Dieter, W., Lumpp, J.: "A user-level checkpointing library for POSIX threads programs"; Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing; (FTCS'99); 224-227; IEEE Computer Society, 1999.

[Fu and Ding 2010] Fu, H., Ding, Y.: "Using Redundant Threads for Fault Tolerance of OpenMP Programs"; Proceedings of the 2010 International Conference on Information Science and Applications; (ICISA'10); 1-8; 2010.

[HDF5] HDF5 website (last accessed: March 2014): http://www.hdfgroup.org/HDF5/.

[NAS NPB] NAS Parallel Benchmarks (last accessed: March 2014): http://www.nas.nasa.gov/publications/npb.html.

[OpenMP] OpenMP website (last accessed: March 2014): http://openmp.org/.

[Prvulovic et al. 2002] Prvulovic, M., Z., Z., Torrellas, J.: "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors"; Proceedings of the 29th Annual International Symposium of Computer Architecture; (ISCA'02); 111-122; IEEE Computer Society, 2002.

[Rodríguez 2008] Rodríguez, G.: Compiler-assisted checkpointing of message-passing applications in heterogeneous environments; Ph.D. thesis; Universidade da Coruña (2008).

[Rodríguez et al. 2009] Rodríguez, G., Martín, M., González, P., Touriño, J.: "A Heuristic Approach for the Automatic Insertion of Checkpoints in Message-Passing Codes"; Journal of Universal Computer Science; 15 (2009), 14, 2894-2911.

[Rodríguez et al. 2011] Rodríguez, G., Martín, M., González, P., Touriño, J.: "Analysis of Performance-impacting Factors on Checkpointing Frameworks: The CPPC Case Study"; The Computer Journal; 54 (2011), 11, 1821-1837.

[Rodríguez et al. 2010] Rodríguez, G., Martín, M., González, P., Touriño, J., Doallo, R.: "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications"; Concurrency and Computation: Practice and Experience; 22 (2010), 6, 749-766.

[Sorin et al. 2002] Sorin, D., Martin, M., Hill, M., Wood, D.: "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery"; Proceedings of the 29th Annual International Symposium on Computer Architecture; (ISCA'02); 123-134; IEEE Computer Society, 2002.

[Stellner 1996] Stellner, G.: "CoCheck: Checkpointing and process migration for MPI"; Proceedings of the 10th International Parallel Processing Symposium; (IPPS'96);

526-531; IEEE Computer Society, 1996.

[Sunil et al. 2003] Sunil, A., Jungwhan, K., Sagyong, H.: "PC/MPI: Design and Implementation of a Portable MPI Checkpointer"; Proceedings of the 10th European PVM/MPI Users' Group Meeting; volume 2840 of Lecture Notes in Computer Science; 302-308; Springer Verlag, 2003.

[Tahan and Shawky 2012] Tahan, O., Shawky, M.: "Using dynamic task level redundancy for OpenMP fault tolerance"; Proceedings of the 25th International Conference on Architecture of Computing Systems; (ARCS'12); 25-36; Springer Verlag, 2012.

[Walker and Dongarra 1996] Walker, D., Dongarra, J.: "MPI: a standard message passing interface"; Supercomputer; 12 (1996), 56-68.

[Woo et al. 2004] Woo, N., Jung, H., Yeom, H., Park, T., Park, H.: "MPICH-GF: Transparent checkpointing and rollback-recovery for grid-enabled MPI processes"; IEICE Transactions on Information and Systems; 87 (2004), 7, 1820-1828.