Journal of Universal Computer Science, vol. 15, no. 14 (2009), 2894-2911
submitted: 29/5/09, accepted: 30/7/09, appeared: 1/8/09 © J.UCS

A Heuristic Approach for the Automatic Insertion of Checkpoints in Message-Passing Codes

Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño
(Computer Architecture Group, University of A Coruña, Spain
{grodriguez,mariam,pglez,juan}@udc.es)

Abstract: Checkpointing tools may be typically implemented at two different abstraction levels: at the system level or at the application level. The latter has become a more popular alternative due to its flexibility and the possibility of operating in different environments. However, application-level checkpointing tools often require the user to manually insert checkpoints in order to ensure that certain requirements are met (e.g. forcing checkpoints to be taken at the user code and not inside kernel routines). The approach presented in this work is twofold. First, a spatial coordination protocol for checkpointing parallel SPMD applications is proposed, based on forcing checkpoints to be taken at the same places in the application code by all processes. Thus, global consistency is achieved without adding any new runtime communications or piggybacked data, and without the need to use specific fault-tolerant message-passing implementations. Second, the paper also introduces a compilation technique for the automatic insertion of checkpoints using the spatial coordination protocol, based on a static analysis of communications and a heuristic analysis of computational load. These analyses can also be used to achieve automatic checkpoint insertion in approaches based on classical protocols, such as uncoordinated checkpointing or distributed snapshots.
Key Words: Fault tolerance, checkpointing, parallel programming, message-passing, compiler-support
Category: C.4, D.1.3

1 Introduction

The basic difference between sequential and parallel applications in terms of failure recovery is the existence of dependencies imposed by interprocess communications. If a checkpoint is placed in the code between two matching communication statements, an inconsistency would occur upon recovery, since the first one will not be executed. Several solutions have been proposed to ensure the consistency of the checkpointing scheme. Uncoordinated checkpointing [Elnozahy et al. 2002] presents a good failure-free performance, but is susceptible to the domino effect [Randell 1975] and, since processes checkpoint independently, there exists the possibility of creating useless checkpoints. Blocking coordinated checkpointing [Tamir and Sequir 1984] avoids the domino effect, but may lead to significant overheads. In order to reduce the effects of coordination, non-blocking coordinated schemes were proposed, the most studied to date being the *distributed snapshots* protocol [Chandy and Lamport 1985].

It would be desirable to retain the efficiency and scalability of uncoordinated checkpointing while, at the same time, guaranteeing the execution progress in the presence of failures. One way to achieve this goal in SPMD codes is to ensure that all checkpoints are created not at the same time, but at the same code locations by all processes. This technique, which we call *spatial coordination*, is further described in the next section.

This paper describes compilation techniques for the automatic insertion of checkpoints in SPMD applications using the spatial coordination protocol, which is formalized in [Section 2]. The process of checkpoint insertion is divided in two fundamental steps: a static analysis of communications, required in order to detect safe points, described in [Section 3]; and a heuristic approach to estimate the computational load of codes, that determines which of the detected safe points are adequate checkpoint locations, covered in [Section 4]. [Section 5] presents the experimental results obtained by the implementation of the presented techniques, evidencing the usability of spatial coordination for a wide range of applications. The related work is discussed in [Section 6], and [Section 7] concludes the paper.

2 Spatial coordination protocol

The proposal of spatially coordinated checkpointing implies identifying, at compile time, code locations in an SPMD message-passing program where it is guaranteed that neither in-transit nor inconsistent messages exist. These code locations are called *safe points*. An example is shown in [Fig. 1], where $c_{i,x}$ represents the x-th checkpoint in process p_i, and each m_k represents a communication between two processes.

Given two checkpoints, $c_{i,x}$ and $c_{j,y}$, let us define the *consistency* relation, $c_{i,x} \sim c_{j,y}$, as a binary relation that is true whenever there are neither in-transit, nor inconsistent messages exchanged by processes p_i and p_j between their x-th and y-th checkpoints, respectively.

Let us assume that:

- All checkpoints are placed at safe points.
- All processes take the same number of checkpoints at the same safe points in the code.

Then, by the definition of safe point, the x-th checkpoint at process p_i will be consistent with the x-th checkpoint at process p_j:

$$\forall i, j, x : c_{i,x} \sim c_{j,x} \tag{1}$$

Let us define the set of all checkpoints taken by each process p_i as $\Omega(p_i)$. Finding a valid recovery line is as simple as finding the checkpoint with index x that verifies:

$$\forall i : c_{i,x} \in \Omega(p_i) \wedge \nexists y / \left(\forall j : c_{j,y} \in \Omega(p_j) \wedge (y > x)\right) \tag{2}$$

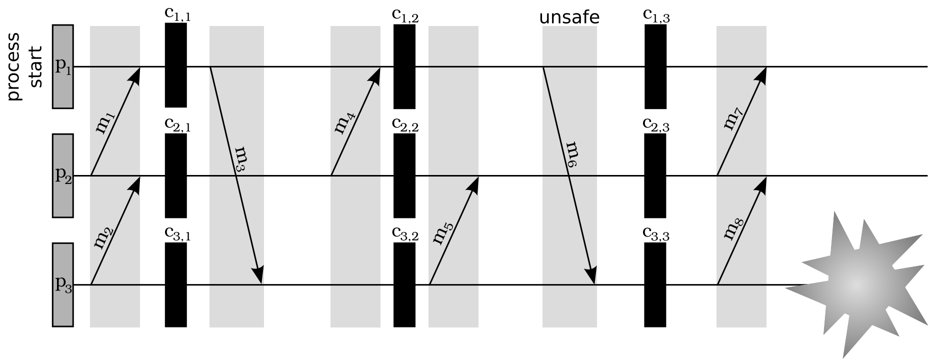

Figure 1: *Spatially coordinated checkpointing*

That is, a checkpoint with index x exists in the set of checkpoints available for recovering all processes, and it is the greatest index that fulfills that condition. By [Eq. (1)], the set $\{c_{i,x},\ i = 1, \ldots, n\}$ forms a valid recovery line for all processes, being n the number of processes executing the parallel application. According to [Eq. (2)], it is also the most recent one. The garbage collection algorithm used to delete useless recovery information is equally simple. Processes can periodically communicate, in an asynchronous way, their most recently created checkpoint index. By simply storing the minimum received value for all processes, a process p_i may determine that any checkpoint with an index lower than the minimum will not be part of any valid recovery line in the future.

This approach has several advantages: (1) it rules out both the possibility of the domino effect, and the creation of useless checkpoints, without the need for any specific protocol at runtime; (2) checkpoints are taken independently by each process, without any added communications, thus in a completely scalable way; and (3) no assumptions are made about the properties of the communications channel, which could be unreliable and/or non-FIFO without affecting the protocol. In order to achieve point (3), approaches based on runtime coordination resort to techniques like piggybacking, which causes an unacceptably high overhead in communication-intensive codes, as stated in [Schulz et al. 2008].

3 Communication analysis

In order to automatically find safe points in the code, communication statements must be analyzed by means of a send-receive matching. The proposed approach is similar to a static simulation of the program execution, and focuses on MPI codes, although it would be easily adaptable for other message-passing libraries. Two communications are considered to match if the following conditions hold:

1. Their sets of sources/destinations are the same: if process p_i executes the send statement using process p_j as destination, then p_j executes the receive statement using p_i as source.

2. Their tags are the same or the receive uses `MPI_ANY_TAG`.

In order to statically compare source/destination pairs and tags, constant propagation and folding must be carried out to determine their literal values during the execution. Since propagating and folding all application constants may prove to be computationally intensive, this operation is restricted to the set of variables that are involved in the computation of parameters affecting communications. This set of *communication-relevant* variables is recursively calculated as:

1. Variables directly used in tags or source/destination parameters.

2. Variables on which a communication-relevant variable depends. Note that this includes the set of variables appearing in conditional expressions that control modifications to other communication-relevant variables.

In order to optimize this process, only statements that modify communication-relevant variables are analyzed for constant folding, since any other does not affect communications. Some communication-relevant variables are multivalued, that is, they potentially have a different value for each process. Thus, the number of processes involved in the execution of the code must be known beforehand to perform the constant folding. For instance, many applications are written in a scalable fashion, using code that dynamically calculates neighbor processes in the communication topology. These calculations are affected by the total number of processes involved in the parallel execution. Thus, values for the variables that contain the number of processes and the rank of a process may be calculated. Note that the latter is multivalued $(0, \ldots, n-1)$. Expressions derived from multivalued variables are, in the general case, multivalued themselves. The constant folding and propagation may be performed together with the communication matching in the same compiler pass.

For keeping track of the communications status, a buffer object is used, starting out empty. The analysis begins at the application's entry point. Statements that are neither control flow- nor communication-related are ignored. Each time a new communication is found, it is first matched against existing ones in the buffer. If a compatible match is not found, the communication is added to the buffer and the analysis continues. If a match is found, both statements are considered linked and removed from the buffer, except when matching non-blocking sends and receives. In this case, they remain in the buffer in an "unwaited" status until a matching wait is found.

A statement in the application code will be considered a safe point if, and only if, the buffer is completely empty when the analysis reaches that statement. An empty buffer implies that no pending communications have been issued, and therefore an in-transit or inconsistent message may not exist at that point.

The following subsections deal with particular aspects of the communication analysis, such as analyzing conditional expressions, procedure calls, collective communications and the limitations of this analysis.

3.1 Conditional statements

When a conditional statement is reached, each conditional branch is independently analyzed using a newly created buffer. The reason for doing this is that no correct partial order for the statements in conditional branches may be established. Communications found inside the conditional construct are added to their corresponding buffer as usual, with an important difference: each one is marked as being executed only if the expression controlling the execution of its conditional path holds. When both branches have been fully analyzed, there will be three different buffers: one for each of the analyzed conditional branches plus the original one (although this is generalizable to conditionals with multiple branches). All three buffers need to be merged into a single one representing the state of the communications buffer after executing the conditional construct. For this purpose, a binary operation ($\amalg$) for merging two buffers B_1 and B_2 is defined, consisting of two fundamental phases:

1. *Redundant* communications are removed from B_1 and B_2. These are communications that are executed in incompatible conditional paths (e.g. on different branches of the same conditional statement), but match the same set of statements according to their source/destination and tag. If these redundant communications were not identified, the algorithm would yield incorrect results, since only one of the equivalent communications would be correctly matched. Redundant communications are substituted by a single representative one, that will be matched at the same point in the code where each of the redundant ones would during execution. This enables consistent discovery of safe points in the presence of redundant communication statements.

2. Communications in B_2 are orderly injected into B_1. Matches are dealt with as when analyzing regular sections of code. Non-matching communications are added to B_1.

Using the merge operation thusly defined, the resulting buffer from the analysis of the conditional block is calculated as:

$$B_o \amalg (B_t \amalg B_f) \tag{3}$$

where B_o represents the original communications buffer (i.e. before reaching the conditional statement), while B_t and B_f are the buffers used for analyzing the true and false clauses of the conditional, respectively. The reason for first merging the buffers obtained from analyzing the conditional branches (according to [Eq. (3)]) is that communications in different conditional paths often match between themselves.

When this operation ends, the resulting buffer represents the communications state of a process that, starting in a state represented by B_o, executes the block of code associated with the conditional statement.

The identification of safe points while analyzing a conditional is deferred until the merging of the buffers is complete. This avoids the identification of false safe points caused by using different independent buffers to analyze each conditional branch.

Note that for some conditionals the control expression may be statically determined during the constant folding. In these cases, only the appropriate conditional branch is analyzed. Situations in which the control expression is multivalued are also detected, and dealt with accordingly.

3.2 Procedure calls

When a procedure call is found, the ongoing analysis is stopped while the code of the called procedure is processed using the same communications buffer. However, communications issued inside the procedure are also cached separately for optimization purposes. If the procedure does not modify any communication-relevant variable, cached results can be reused when a new call to the same procedure is found, without analyzing the procedure code, but just symbolically analyzing their tags and source/destination parameters again.

This optimization is employed whenever possible, thus avoiding the repeated analysis of a procedure each time a call to it is found. If the procedure modifies any communication-relevant variable, then it is analyzed each time a call is found, since these modifications have to be tracked and applied to guarantee correct constant folding.

3.3 Collective communications

In SPMD applications, the usual way of coding collective communications is to make all involved processes execute a single, non-conditional line of code. Before and after the execution of that line the collective communication does not affect the consistency of communications. However, in the general case, codes may contain a collective communication spread across several conditional paths. In this case, all its parts are detected as redundant and it is ensured that checkpoint placement does not generate restart inconsistencies.

3.4 Matching ambiguous communications

Some parallel applications present irregular communication patterns (those where tags and/or source/destination parameters are derived from the input data) or nondeterministic communications (which use wildcard receives). In these situations, the information available at compile time may not be enough to completely determine how the communications will play out during runtime. To ensure the correctness of the results, a conservative solution is used. This approach is based on considering any of the potential matches, deferring the match to the latest possible one in the code. The correctness of the results is guaranteed, although some actual safe points may not be considered as such for the sake of consistency.

3.5 Limitations

When dealing with applications featuring ambiguous communications, as described in the previous subsection, the proposed solution might be unable to find suitable safe points. This makes it impossible to deploy checkpoints at all the locations selected by the checkpoint insertion analysis described in the next section. However, such situations are uncommon, as shown in the experimental assessment of the approach in [Section 5]. Despite some of the analyzed applications featuring ambiguous communications, suitable safe points were found for all proposed checkpoint locations, proving the usability of the approach for a wide range of applications.

4 Heuristic computational load analysis

In order to guarantee execution progress in the presence of failures, checkpoints need to be inserted at places in the code that perform the core of the computation, and thus take the longest time to execute. However, it is not possible to accurately predict the execution time of a section of code without precise knowledge of the hardware executing it. To overcome this issue heuristic analyses are employed, using computational metrics to discover critical sections of code.

The first step is to discard any code that is not nested inside a loop. This decision is supported by the fact that any block of code which takes enough time to run as to grant checkpointing during its execution is necessarily a loop, with possibly nested loops in its body. After the isolation of the loop nests in the code, the compiler proceeds to rank all such nests in order of estimated computational load. After this step, the compiler dynamically selects a load threshold. Loop nests above the threshold are checkpointed, while those below it are deemed not relevant enough to guarantee checkpoint insertion. The decision to dynamically select the threshold, as opposed to defining it as a constant in the compiler code, responds to a desire to take into account how the total computational load of the application is divided into its loop nests before selecting checkpoint locations.

The actual heuristic computational metric used is derived from two simpler ones: the number of statements inside the loop and the number of variables these statements access. The combination of both approximates the associated computational load better than any of them. For instance, a loop may have many statements that access mostly constants. These are not good candidates for checkpointing, since they typically perform variable initializations. Conversely, a loop might execute a low number of statements which perform costly computations. These are typically characterized for accessing a large amount of variables. Note that the obtention of both metrics must be done in an interprocedural way: the call graph of the application must be navigated, and the characteristics of the contained subroutine calls taken into account for the calculation of the number of statements and variable accesses contained in the loop code.

Let l be a loop in L, the loop population of the application P, and s and a two functions that give the number of statements and variable accesses in a given block of code, respectively. Let us define $S(l) = s(l)/s(P)$ and $A(l) = a(l)/a(P)$ the total proportion of statements and accesses, in that order, that exist inside a given loop l. The heuristic computational load value associated to each loop l is calculated as:

$$h(l) = -log(S(l) \cdot A(l)) \tag{4}$$

[Eq. (4)] multiplies $S(l)$ and $A(l)$ to ensure that the product is bigger for loops that are significant for both metrics. It applies a logarithm to make variations smoother. Finally, it takes the negative of the value to make $h(l)$ strictly positive. Thus, the lower the value the higher the computing time estimated for that loop. In order to select the best candidates for checkpoint insertion, the compiler ranks loops in L attending to $h(l)$ and applies thresholding methods [Sezgin and Sankur 2004]. After exploring several possibilities, a two-step method has been adopted. First, a "shape-based" approach is used to select the subset of the time-consuming loop nests in the application. The second step restricts this selection by means of a "cluster-based" technique.

[Fig. 2] shows the proposed method applied to the BT application of the MPI version of the NAS Parallel Benchmarks (NPB) [NAS NPB]. The loops have been arranged in an ascending order from left to right. In the lower left corner, l_0 corresponds to the loop in the application with the biggest estimated computational load. The darker part on the left corresponds to the subset of loops selected by the first thresholding step. The darker rectangle on the right graphically depicts the processing of this subset by the second thresholding method. In the first step, using the so-called "triangle method", loops in L are divided into two classes: time-consuming loops and negligible loops. Conceptually, the triangle method consists in: (a) drawing a line between the first and last values

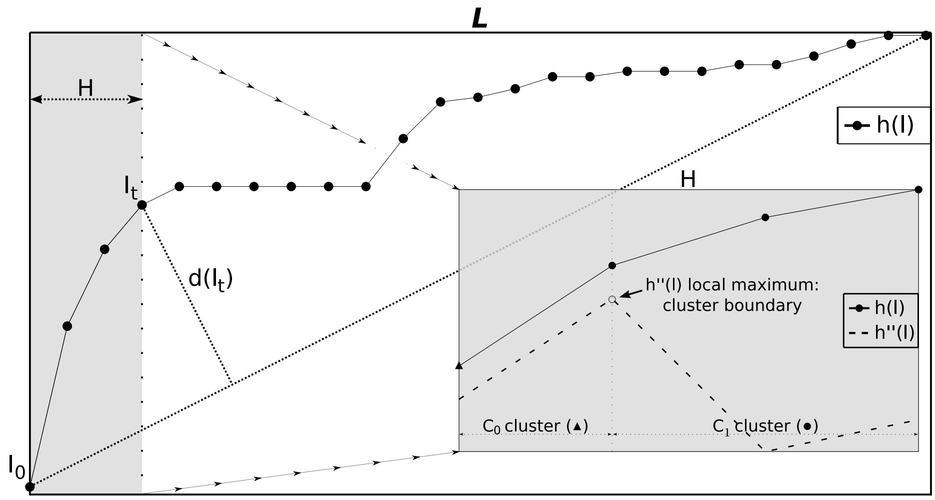

Figure 2: *Thresholding method for checkpoint insertion in the NPB BT application. Loop nests in the loop population L are ordered in the x axis according to their $h(l)$ value, represented in the y axis. The darker section on the lower right is a detail of the subset H of loops in L selected by the first thresholding step, and represents both $h(l)$ and $h''(l)$ in the y axis.*

of $h(l)$; (b) calculating the perpendicular distance $d(l)$ to this line for all $l \in L$; and (c) calculating a threshold value l_t such that $d(l_t) = max\{d(l)\}$. This first step selects a subset of the loop population, referred to as H, as time-consuming loops candidates for checkpoint insertion. The triangle method was originally developed in the context of image processing [Zack et al. 1977], and appears to be especially effective when there is a narrow peak in the histogram, which is often the case for the proposed $h(l)$ function in real applications.

However, this first threshold selects more loops than would be desirable for checkpoint insertion. The second step of the proposed algorithm refines this selection using a cluster-based thresholding algorithm. Loops in H with similar associated costs are grouped into clusters, C_i, built by selecting the local maximums of the second derivative of $h(l)$ as partitioning limits. Since $h(l)$ is monotonically increasing, local maximums in $h''(l)$ represent inflection points at which $h(l)$ begins to change more smoothly. For an application with k clusters, the method calculates a threshold value t such that:

$$\sum_{i=0}^{t} h(l_{i+1}^0) - h(l_i^0) > \sum_{i=t+1}^{k-1} h(l_{i+1}^0) - h(l_i^0) \qquad (5)$$

Loop #	**File**	**Line**	**Statements**	**Accesses**	**h(l)**	**runtime (s)**
1	`bt.f`	179	2075	5547	0.7755	143.12
2	`exact_rhs.f`	24	107	490	3.1149	0.14
3	`initialize.f`	44	30	132	4.2368	$8.44 \cdot 10^{-2}$
4	`error.f`	25	19	47	4.8837	$1.70 \cdot 10^{-2}$
Total program statements					5084	
Total variable accesses					13502	

Table 1: *Loops selected by the shape-based threshold for NPB BT*

where l_i^0 denotes the first loop in C_i. This method selects for checkpoint insertion all the loops inside the clusters C_i such that $i \leq t$. In the example, the first step of the thresholding process selects a subset of four loop nests, out of the total 25 in the application, as candidates for checkpoint insertion. The values of $h(l)$ for these loops are detailed in [Tab. 1], as well as the total number of statements and variable accesses in the application, shown for reference, and the real time it takes for an execution of this application to run each of the considered loops. These times were measured executing the BT class A code on 32 Itanium 2 cores. As can be seen, significant differences in $h(l)$ for different loops respond to significant differences in loop runtimes. Thus, the heuristic works reasonably well for estimating computational load differences. The second thresholding step groups all loops into two clusters, delimited by the single maximum in $h''(l)$. Applying [Eq. (5)], the algorithm determines that the loops in C_0 are responsible for 56.94% of the total variation of $h(l)$ in the H subset, and therefore selects only loop #1 for checkpoint insertion. This loop is the main computational loop in BT, and is the place where a manual checkpoint should be inserted.

Once identified the loops in which checkpoints are to be inserted, the results of the communication analysis described in the previous section are used to insert a checkpoint at the first available safe point in each selected loop nest. This approach can be also used to detect adequate checkpoint locations when using other application-level checkpointing approaches (e.g. uncoordinated, distributed snapshots, etc.). Experimental results assessing the effectiveness of the proposed method are detailed in the next section.

5 Experimental results

The communication analysis and the checkpoint insertion algorithm described in the previous sections have been implemented into CPPC [Rodríguez et al. 2010], an application-level checkpointing framework. CPPC focuses on achieving the portability of the generated state files, making it possible to restart applications on architectures and/or OS different from those which originally gener-

<table>
<tr><th>Application</th><th>LOCs</th><th>#L</th><th colspan="2">Manual checkpoints (file:line)</th></tr>
<tr><td>BT</td><td>3650</td><td>25</td><td>1</td><td><code>bt.f</code>:179</td></tr>
<tr><td>CG</td><td>1044</td><td>13</td><td>1</td><td><code>cg.f</code>:441</td></tr>
<tr><td>EP</td><td>180</td><td>4</td><td>1</td><td><code>ep.f</code>:189</td></tr>
<tr><td>FT</td><td>1269</td><td>20</td><td>1</td><td><code>ft.f</code>:159</td></tr>
<tr><td>IS</td><td>672</td><td>6</td><td>1</td><td><code>is.c</code>:976</td></tr>
<tr><td>LU</td><td>3086</td><td>35</td><td>1</td><td><code>ssor.f</code>:78</td></tr>
<tr><td>MG</td><td>1618</td><td>12</td><td>1</td><td><code>mg.f</code>:245</td></tr>
<tr><td>SP</td><td>3148</td><td>25</td><td>1</td><td><code>sp.f</code>::150</td></tr>
<tr><td>CalcuNetw</td><td>810</td><td>14</td><td>1</td><td><code>lee.c</code>:320</td></tr>
<tr><td>Fekete</td><td>182</td><td>6</td><td>1</td><td><code>fekete.f</code>:98</td></tr>
<tr><td rowspan="2">DBEM</td><td rowspan="2">12533</td><td rowspan="2">92</td><td>1</td><td><code>frmtrx.f</code>:130</td></tr>
<tr><td>2</td><td><code>solver.f</code>:240</td></tr>
<tr><td>STEM-II</td><td>6506</td><td>24</td><td>1</td><td><code>main.f</code>:437</td></tr>
</table>

Table 2: *Summary of test applications*

ated the files. In order to improve efficiency, it only stores live variables and uses a spatially coordinated approach. The framework is made up of a library and a compiler. The CPPC library contains checkpointing routines. The CPPC source-to-source compiler helps achieve transparency by relieving the user from time-consuming tasks, such as data flow analyses and adding instrumentation code. When extended with the automatic insertion of checkpoints presented in this paper, the CPPC compiler provides a fully transparent approach to checkpointing.

In order to conduct the experimental evaluation of the automatic insertion of checkpoints, twelve applications were selected. The eight applications in the NPB-MPI v3.1 benchmarks [NAS NPB]; two scientific applications in use in the Supercomputing Center of Galicia (CESGA), *CalcuNetw* [Mouriño et al. 2003] and *Fekete* [Bendito et al. 2007]; and two additional applications called *DBEM* [González et al. 2000] and *STEM-II* [Martín et al. 2003], included to test the tool with large codes. The NPB are well-known and widespread applications that provide a de-facto test suite. CalcuNetw computes some characterization measurements in a given network, consisting of a set of nodes or vertices joined together in pairs by links or edges, and compares it with a number of random networks specified by the user. This is the only sequential application, and thus no communication analysis is necessary. It is included to assess the behavior of the analyses in this type of programs. Fekete determines the position of a certain number of points on a 2-dimensional sphere such that the potential energy produced by the interaction of these points is minimum. This is the 7th

Application	Automatic chkpts. (file:line)		Communication analysis time	Checkpoint insertion time
BT	1	`bt.f`:179	2.449 s.	0.851 s.
CG	1	`cg.f`:441	0.636 s.	0.080 s.
EP	1	`ep.f`:189	0.078 s.	0.011 s.
FT	1	`ft.f`:159	0.837 s.	0.273 s.
IS	1	`is.c`:425	1.679 s.	0.048 s.
	2	`is.c`:976		
	3	`is.c`:396		
LU	1	`ssor.f`:78	1.304 s.	0.388 s.
MG	1	`mg.f`:245	7.707 s.	0.317 s.
SP	1	`sp.f`:150	1.756 s.	0.783 s.
	2	`exact_rhs.f`:23		
	3	`initialize.f`:45		
	4	`error.f`:26		
CalcuNetw	1	`lee.c`:320	0 s.	0.198 s.
Fekete	1	`fekete.f`:98	0.070 s.	0.015 s.
DBEM	1	`frmtrx.f`:130	15.233 s.	20.334 s.
	2	`solver.f`:240		
	3	`stress.f`:37		
	4	`gausspt.f`:25		
STEM-II	1	`main.f`:437	3.553 s.	1.448 s.

Table 3: *Compilation techniques results*

Loop #	File	Line	Statements	Accesses	h(l)
*1	`is.c`	425	90	154	0.6570
*2	`is.c`	976	56	39	1.4595
*3	`is.c`	396	36	59	1.4716
4	`is.c`	387	23	38	1.8573
5	`is.c`	882	16	10	2.5947
Total program statements					242
Total variable accesses					260

Table 4: *Loops selected by the shape-based threshold for NPB IS*

of the Smale's problems [Smale 1998]. DBEM performs a crack growth analysis that leads to a large number of discretized equations. It solves the resulting dense linear system using the GMRES iterative method. STEM-II is an air quality sim-

Loop #	File	Line	Statements	Accesses	h(l)
*1	`sp.f`	150	927	1783	1.0077
*2	`exact_rhs.f`	23	107	490	2.5044
*3	`initialize.f`	45	30	132	3.6262
*4	`error.f`	26	19	47	4.2731
5	`initialize.f`	103	12	40	4.5427
Total program statements					2801
Total variable accesses					6009

Table 5: *Loops selected by the shape-based threshold for NPB SP*

Loop #	File	Line	Statements	Accesses	h(l)
*1	`frmtrx.f`	130	1924	2869	0.5017
*2	`solver.f`	240	333	438	2.0797
*3	`stress.f`	37	214	426	2.2838
*4	`gausspt.f`	25	182	457	2.3236
5	`bcs.f`	123	135	237	2.7385
6	`prpdir.f`	34	84	180	3.0640
7	`prpgtn.f`	273	89	66	3.4747
8	`sif.f`	43	52	95	3.5499
9	`linsyslp.f`	473	60	69	3.6266
10	`prpgtn.f`	82	53	41	3.9065
Total program statements					3330
Total variable accesses					5262

Table 6: *Loops selected by the shape-based threshold for DBEM*

ulation, used to know in advance how the meteorological conditions, obtained from a meteorological prediction model, would affect the emissions of pollutants by the power plant of As Pontes (A Coruña, Spain) in order to fulfill EU regulations. [Tab. 2] shows a summary of the characteristics of the test applications, including lines of code (LOCs), the number of loop nests in the code ($\#L$), and the place (file and line number of the source code) where each checkpoint was manually inserted by the authors during the assessment of the test applications.

Qualitatively, the results of the communication analysis for the test applications were always correct. Even for NPB IS, which presents an irregular communication pattern, the analysis classified lines in the code as safe or unsafe points appropriately. Regarding the automatic checkpoint insertion, [Tab. 3] details the places selected by the proposed heuristic. It can be observed that, compared to the manually inserted checkpoints in [Tab. 2], some extra checkpoints are inserted in applications with similar heuristic values for both time-consuming loops and negligible ones, but a time-consuming loop is never left uncheckpointed. Thus, the proposed approach avoids significant loss of work. The table also details processing times, measured in a desktop computer, with an Intel Core2 Duo at 3 GHz and 1 MB of RAM. Although the number of applications is insufficient to develop a complete mathematical model of execution times, tendencies can be inferred. The communication analysis tends to $O(LOCs^2)$. The checkpoint insertion analysis does not depend on the LOCs of the application, but rather on the number of loop nests, being $O(\#L^2)$. As shown in [Tab. 3], the time spent in the analyses is acceptably low for all test applications.

Analyzing the inserted checkpoints, for three out of the total twelve applications (IS, SP and DBEM) the compiler behavior does not match the manually inserted checkpoints. In the NPB IS application, it selects two loops in addition to the main computational loop. This conservative behavior is caused by the small size of the application together with the fact that its loops have very similar sizes. In fact, this is the application with the smallest total variation in $h(l)$, and the one in which $h(l)$ most closely resembles a straight line. This causes the first step of the thresholding process to select a high number of loop nests as candidates for checkpoint insertion, five out of the total six in IS. [Tab. 4] details the loops selected by the shape-based thresholding step, being marked with an asterisk those ultimately selected for checkpointing by the cluster-based thresholding step. In NPB SP, the algorithm inserts four checkpoints, three of them conservatively selected, as can be seen in [Tab. 5]. Conservative checkpoints are also added for DBEM, as detailed in [Tab. 6]. Loop #1 has such a high associated cost that it equalizes the $h(l)$ range for the other loops. Thus, the cost function associated to loop #2 is very similar to other non-time-consuming ones. This results in two extra loops being checkpointed. For comparison purposes, [Tab. 7] and [Tab. 8] present the loops in the H subset for NPB FT and NPB LU, respectively, two of the applications for which the automatically inserted checkpoints match the manually inserted ones.

6 Related work

Many checkpointing approaches in the literature perform structural analyses and source code modifications to instrument checkpointing insertion. Most of them, however, leave the insertion of checkpoints to be manually done by the user, while automatically performing the remaining instrumentation (variable storage, recovery, control flow, etc.). Porch [Ramkumar and Strumpen 1997] and C^3 [Bronevetsky et al. 2003] require that the user inserts checkpoint calls in the code. These calls will only trigger an actual checkpoint according to a frequency timer. These “potential checkpoints” were originally introduced by CATCH

Loop #	**File**	**Line**	**Statements**	**Accesses**	**h(l)**
*1	`ft.f`	159	189	337	1.2005
2	`ft.f`	676	9	24	3.5398
3	`ft.f`	689	9	24	3.5398
4	`ft.f`	702	9	24	3.5398
5	`ft.f`	1016	7	9	4.0750
6	`ft.f`	271	3	7	4.5521
Total program statements					743
Total variable accesses					1261

Table 7: *Loops selected by the shape-based threshold for NPB FT*

Loop #	**File**	**Line**	**Statements**	**Accesses**	**h(l)**
*1	`ssor.f`	78	452	2251	1.0466
2	`erhs.f`	383	43	151	3.4828
3	`erhs.f`	118	33	116	3.7122
Total program statements					1961
Total variable accesses					7415

Table 8: *Loops selected by the shape-based threshold for NPB LU*

GCC [Li et al. 1994]. This tool for sequential applications also automated their insertion by introducing a potential checkpoint at the beginning of subroutines and at the first line inside a loop. This checkpoint placement guaranteed, in the general case, that potential checkpoint calls would be executed often enough so as to provide a checkpointing frequency reasonably close to the desired one. This approach cannot be followed when using a spatial coordination protocol. In this situation, checkpointing frequencies are not defined in temporal terms, due to the need to statically coordinate all processes independently of how long they take to progress through the application's execution. Instead of statically detecting safe points, C^3 employs a coordinated protocol based on piggybacking information into sent messages. Thus, every message being sent has to be intercepted and modified. This introduces significant overheads when dealing with applications with intensive collective communications, since each collective message is translated to several point-to-point ones.

For checkpointing schemes that do not use runtime coordination, such as our proposal, checkpoints have to be inserted at places where the global consistency of the approach is guaranteed. When working with implicitly parallel languages, the compiler is able to use the native constructions for parallelism to extract information about safe points for checkpointing. Such is the case in [Choi and Deitz 2002], a work for the ZPL language where safe communication-free checkpoint ranges are detected, and a checkpoint is inserted at the single location in the range with the fewest live variables.

When working with explicit communication interfaces, such as MPI, the communication pattern is user-defined. The compiler has to interpret the communication statements inserted in the code, and perform a communication matching to find safe locations for checkpointing. Some analysis models have been developed with the goal of testing the correctness of the communication pattern in an application. However, pursuing correctness verification is a tougher problem than finding safe regions in the code, and these models typically present strong limitations, such as not being able to deal with non-blocking communications or wildcard receives (`MPI_ANY_SOURCE` and `MPI_ANY_TAG`) [Siegel and Avrunin 2005].

Performing static communication analyses typically involves the use of control flow graphs. For instance, in [Shires et al. 1999] this concept is extended for MPI with the introduction of communication edges that connect communication nodes. Since correctness tests are not one of their goals, they assume that the communications are correct and conservatively represent indeterminacies by drawing all possible communication edges between the nodes involved. Using this approach, safe points could be located by identifying nodes that have no communication edges connecting one of its ancestors with one of its successors. Since our goal is not to obtain a representation of the communication pattern in the application, but rather to simply categorize points in the code as either safe or non-safe, the graph may be omitted, using instead the communication buffer object described in [Section 3] to represent pending communications at each point in the code. Not building the graph has several advantages in terms of efficiency. Mainly, it minimizes memory consumption and does not require the analysis of the graph resulting from the compiler pass.

With regards to the automatic checkpoint insertion, some theoretical approaches calculating the optimal mathematical solution to the checkpoint placement problem exist [Toueg and Babaoğlu 1984, Vaidya 1997]. However, these works assume that involved parameters such as underlying hardware, execution time, etc. are known in advance. This is not the case under the CPPC framework, where the execution environment is not assumed to be fixed due to its inherent characteristic of portability.

7 Concluding remarks and future work

The main contributions of this work are the introduction of the spatial coordination protocol for checkpointing message-passing applications, and the development of an algorithm to automate checkpoint insertion using this protocol. This technique consists in the automatic identification of safe points by means of a communication analysis, followed by the recognition of computation-intensive loops based on the analysis of computational load metrics.

The heuristic algorithm has been experimentally tested, demonstrating the validity of the approach for SPMD codes. It correctly selected adequate checkpoints for all test applications, in places that guaranteed the progress of the execution in the presence of failures. We are currently exploring ways to improve the computational load metrics in order to enhance the efficiency of the resulting codes by reducing the amount of conservatively-inserted checkpoints.

This algorithm has been integrated into the CPPC checkpointing framework since version 0.7, where it helps achieve a completely transparent application-level operation. CPPC is publicly available at `http://cppc.des.udc.es` under GPL license.

The checkpoint insertion phase of the presented algorithm can be straightforwardly used to insert checkpoints when working with other consistency schemes, such as uncoordinated checkpointing, the distributed snapshots algorithm or in message-logging approaches.

Acknowledgments

This research was supported by the Ministry of Science and Innovation of Spain and FEDER funds of the European Union (Project TIN-2007-67537-C03-02). We gratefully thank the Supercomputing Center of Galicia (CESGA) and the VARIDIS research group of the Universitat Politècnica de Catalunya for providing access to the CalcuNetw and Fekete applications.

References

[Bendito et al. 2007] Bendito, E., Carmona, A., Encinas, A.M., Gesto, J.M.: "Estimation of Fekete Points"; J. Comput. Phys., 225, 2 (2007), 2354-2376.

[Bronevetsky et al. 2003] Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: "Automated Application-Level Checkpointing of MPI Programs"; Proc. ACM SIGPLAN 2003 Symp. Principles Pract. Parall. Prog. (Jun 2003), 84-94.

[Chandy and Lamport 1985] Chandy, K.M., Lamport, L.: "Distributed Snapshots: Determining Global States of Distributed Systems"; ACM T. Comput. Syst., 3, 1 (1985), 63-75.

[Choi and Deitz 2002] Choi, S.E., Deitz, S.J.: "Compiler Support for Automatic Checkpointing"; Proc. 16^{th} Ann. Intl. Symp. High Performance Comp. Syst. Apps. (Jun 2002), 213-220.

[Elnozahy et al. 2002] Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: "A Survey of Rollback-Recovery Protocols in Message-Passing Systems"; ACM Comput. Surv., 34, 3 (2002), 375-408.

[González et al. 2000] González, P., Pena, T.F., Cabaleiro, J.C., Rivera, F.F.: "Dual BEM for Crack Growth Analysis on Distributed-Memory Multiprocessors"; Adv. Eng. Softw., 31, 12 (2000), 921-927.

[Li et al. 1994] Li, C.C.J., Stewart, E.M., Fuchs, W.K.: "Compiler-Assisted Full Checkpointing"; Software Pract. Exper., 24, 10 (1994), 871-886.

[Martín et al. 2003] Martín, M.J., Singh, D.E., Mouriño, J.C., Rivera, F.F., Doallo, R., Bruguera, J.D.: "High Performance Air Pollution Modeling for a Power Plant Environment"; Parallel Comput., 29, 11-12 (2003), 1763-1790.

[Mouriño et al. 2003] Mouriño, J.C., Estrada, E., Gómez, A.: "CalcuNetw: Calculate Measurements in Complex Networks"; Tech. Rep. CESGA-2005-003, Supercomp. Center of Galicia (2005).

[NAS NPB] National Aeronautics and Space Administration: The NAS Parallel Benchmarks (retrieved May 2009). `http://www.nas.nasa.gov/Software/NPB`.

[Ramkumar and Strumpen 1997] Ramkumar, B., Strumpen, V.: "Portable Checkpointing for Heterogeneous Architectures"; Proc. 27^{th} Intl. Symp. Fault-Tolerant Comp. (Jun 1997), 58-67.

[Randell 1975] Randell, B.: "System Structure for Software Fault Tolerance"; IEEE T. Software Eng., 1, 2 (1975), 221-232.

[Rodríguez et al. 2010] Rodríguez, G., Martín, M.J., González, P., Touriño, J., Doallo, R.: "CPPC: A Compiler-Assisted Tool for Portable Checkpointing of Message-Passing Applications"; Concurr. Comp.: Pract. Exp., In Press (2010).

[Schulz et al. 2008] Schulz, M., Bronevetsky, G., de Supinski, B.R.: "On the Performance of Transparent MPI Piggyback Messages"; Lect. Notes Comp. Sci., 5205 (2008), 194-201.

[Sezgin and Sankur 2004] Sezgin, M., Sankur, B.: "Survey over Image Thresholding Techniques and Quantitative Performance Evaluation"; J. Electron. Imaging, 13, 1 (2004), 146-168.

[Shires et al. 1999] Shires, D., Pollock, L., Sprenkle, S.: "Program Flow Graph Construction for Static Analysis of MPI Programs"; Proc. Intl. Conf. Parall. Dist. Processing Techniques Apps. (Jun 1999), 1847-1853.

[Siegel and Avrunin 2005] Siegel, S.F., Avrunin, G.S.: "Modeling Wildcard-Free MPI Programs for Verification"; Proc. ACM SIGPLAN 2005 Symp. Principles Pract. Parall. Prog. (Jun 2005), 95-106.

[Smale 1998] Smale, S.: "Mathematical Problems for the Next Century"; Math. Intell., 20, 2 (1998), 7-15.

[Tamir and Sequir 1984] Tamir, Y., Sequin, C.H.: "Error Recovery in Multicomputers Using Global Checkpoints"; Proc. 13^{th} Intl. Conf. Parall. Proc. (Aug 1984), 32-41.

[Toueg and Babaoğlu 1984] Toueg, S., Babaoğlu, Ö.: "On the Optimum Checkpoint Selection Problem"; SIAM J. Comput., 13, 3 (1984), 630-649.

[Vaidya 1997] Vaidya, N.H.: "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme"; IEEE T. Comput., 46, 8 (1997), 942-947.

[Zack et al. 1977] Zack, G.W., Rogers, W.E., Latt, S.A.: "Automatic measurement of sister chromatid exchange frequency"; J. Histochem. Cytochem., 25, 7 (1977), 741-753.